CAE Specification

Systems Management:
Distributed Software Administration

The Open Group

© *RJanuary 1998, The Open Group*

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.

CAE Specification

Systems Management: Distributed Software Administration

ISBN: 1-85912-149-7

Document Number: C701

Published in the U.K. by The Open Group, RJanuary 1998.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

Chapter	**1**	**Introduction**	**1**
	1.1	Scope	1
	1.2	Dependencies	3
	1.2.1	Features Inherited From POSIX.1	3
	1.2.2	Features Inherited From POSIX.2	3
	1.3	Conformance	4
	1.3.1	Full Conformance	4
	1.3.2	Limited Conformance	4
Chapter	**2**	**Software Structures**	**5**
	2.1	Classes and Attributes	5
	2.2	Software_Collection	7
	2.3	Distribution	9
	2.4	Media	10
	2.5	Installed_Software	11
	2.6	Vendor	12
	2.7	Category	13
	2.8	Software	14
	2.9	Products	16
	2.10	Bundles	20
	2.11	Filesets	23
	2.12	Subproducts	28
	2.13	Software_Files	29
	2.14	Files	31
	2.15	Control_Files	33
Chapter	**3**	**Common Definition for Utilities**	**35**
	3.1	Synopsis	35
	3.2	Description	35
	3.3	Options	35
	3.3.1	Non-interactive Operation	37
	3.4	Operands	38
	3.4.1	Software Specification and Logic	38
	3.4.1.1	Fully-qualified Software_spec	42
	3.4.1.2	Software Compatibility	42
	3.4.2	Source and Target Specification and Logic	42
	3.5	External Influences	44
	3.5.1	Defaults and Options Files	44
	3.5.2	Extended Options	44
	3.5.3	Extended Options Syntax	52
	3.5.3.1	Precedence for Option Specification	54
	3.5.4	Input Files	54

	3.5.5	Access and Concurrency Control	54
	3.5.6	Environment Variables	55
	3.6	External Effects	56
	3.6.1	Control Script Execution and Environment	56
	3.6.1.1	Control Script Stdout and Stderr	57
	3.6.1.2	Control Script Return Code	57
	3.6.2	Asynchronous Events	58
	3.6.3	Stdout	69
	3.6.4	Stderr	69
	3.6.5	Logging	69
	3.7	Extended Description	70
	3.7.1	Selection Phase	70
	3.7.1.1	Starting a Session	70
	3.7.1.2	Specifying Targets	70
	3.7.1.3	Specifying the Source	71
	3.7.1.4	Software Selections	72
	3.7.2	Analysis Phase	72
	3.7.3	Execution Phase	73
	3.7.3.1	Fileset State Transitions	73
	3.8	Exit Status	75
	3.9	Consequences of Errors	75
	3.10	Error Conditions	75
Chapter	**4**	**Software Administration Utilities**	**77**
		swask	78
		swconfig	81
		swcopy	85
		swinstall	90
		swlist	104
		swmodify	108
		swpackage	111
		swremove	115
		swverify	121
Chapter	**5**	**Software Packaging Layout**	**125**
	5.1	Directory Structure	126
	5.1.1	Exported Catalog Structure	126
	5.1.1.1	INDEX File	126
	5.1.1.2	Distribution Files	126
	5.1.1.3	Product Catalog	127
	5.1.1.4	Product Control Files	127
	5.1.1.5	Fileset Control Files	128
	5.1.2	File Storage Structure	128
	5.1.2.1	Control Directory Names	129
	5.2	Software Definition File Format	130
	5.2.1	Software Definition File Syntax	130
	5.2.1.1	Keyword and Attribute Semantics	134
	5.2.1.2	Vendor Defined Keywords and Attributes	135

	5.2.2	Distribution Definition	135
	5.2.3	Media Definition	135
	5.2.4	Installed Software Definition	136
	5.2.5	Vendor Definition	136
	5.2.6	Category Definition	136
	5.2.7	Bundle Definition	137
	5.2.8	Product Definition	138
	5.2.9	Subproduct Definition	139
	5.2.10	Fileset Definition	139
	5.2.11	Control_File Definition	140
	5.2.12	File Definition	141
	5.2.13	Extended Control_File Definitions	142
	5.2.14	Extended File Definitions	143
	5.2.14.1	Directory Mapping	143
	5.2.14.2	Recursive File Definition	143
	5.2.14.3	Explicit File Definition	144
	5.2.14.4	Default Permission Definition	146
	5.2.14.5	Excluding Files	146
	5.2.14.6	Including Files	146
	5.2.15	Space Control_file	146
	5.3	Serial Format and Multiple Media	147
Appendix	**A**	**Sample File Coding**	**149**
	A.1	Defaults File	149
	A.2	Product Specification File	152
	A.3	Software Packaging Layout	154
	A.4	INDEX File	155
	A.5	INFO File	157
	A.6	Control Script	158
	A.7	Patch PSF Example	159
Appendix	**B**	**Background Information**	**161**
	B.1	General	161
	B.1.1	Scope and Purpose	161
	B.1.2	Roles	161
	B.1.3	Tasks	165
	B.1.4	Update Requirements	167
	B.1.5	Patch Requirements	168
	B.1.6	Conformance	169
	B.1.6.1	Implementation Conformance	169
	B.1.6.2	Distribution Conformance	169
	B.2	Software Structures	171
	B.2.1	Classes and Attributes	171
	B.2.2	Software_Collection	176
	B.2.3	Distribution	176
	B.2.4	Media	177
	B.2.5	Installed_Software	177
	B.2.6	Vendor	177

B.2.7 Category ... 178
B.2.8 Software ... 178
B.2.9 Products ... 179
B.2.10 Bundles ... 181
B.2.11 Filesets ... 183
B.2.12 Subproducts ... 183
B.2.13 Software_Files ... 183
B.2.14 Files ... 185
B.2.15 Control Files ... 186
B.3 Common Definitions for Software Administration Utilities ... 187
B.3.1 Synopsis ... 187
B.3.2 Description ... 187
B.3.3 Options ... 187
B.3.3.1 Non-Interactive Operation ... 187
B.3.4 Operands ... 187
B.3.4.1 Software Specification and Logic ... 188
B.3.4.2 Source and Target Specification and Logic ... 190
B.3.5 External Influences ... 191
B.3.5.1 Defaults and Options Files ... 191
B.3.5.2 Extended Options ... 191
B.3.5.3 Extended Options Syntax ... 192
B.3.5.4 Standard Input ... 195
B.3.5.5 Input Files ... 195
B.3.5.6 Access and Concurrency Control ... 195
B.3.6 External Effects ... 196
B.3.6.1 Control Script Execution and Environment ... 196
B.3.6.2 Asynchronous Events ... 200
B.3.6.3 Stdout ... 200
B.3.6.4 Stderr ... 200
B.3.6.5 Logging ... 200
B.3.7 Extended Description ... 201
B.3.7.1 Selection Phase ... 201
B.3.7.2 Analysis Phase ... 201
B.3.7.3 Execution Phase ... 201
B.3.8 Exit Status ... 202
B.3.9 Consequences of Errors ... 202
B.4 Software Administration Utilities ... 203
swask ... 204
swconfig ... 207
swcopy ... 210
swinstall ... 212
swlist ... 221
swmodify ... 226
swpackage ... 228
swremove ... 230
swverify ... 233
B.5 Software Packaging Layout ... 235
B.5.1 Directory Structure ... 236

B.5.2 Software Definition File Format 236
B.5.3 Serial Format and Multiple Media 240

Glossary **243**

Index **253**

List of Figures

B-1 Roles in Software Administration 162
B-2 Example of Software Structure 171
B-3 Software Object Containment 173
B-4 Software Object Inheritance 174
B-5 Fileset State Transitions (Within Distributions) 201
B-6 Fileset State Transitions (Within Installed Software) 202
B-7 Installation State Changes 212
B-8 Order of Install Operations 214
B-9 Order of Remove Operations 230

List of Tables

2-1 Attributes of the Software_Collection Common Class 7
2-2 Attributes of the Distribution Class 9
2-3 Attributes of the Media Class 10
2-4 Attributes of the Installed Software Class 11
2-5 Attributes of the Vendor Class 12
2-6 Attributes of the Category Class 13
2-7 Attributes of the Software Common Class 14
2-8 Attributes of the Product Class 16
2-9 Attributes of the Bundle Class 20
2-10 Attributes of the Fileset Class 23
2-11 Attributes of the Subproduct Class 28
2-12 Attributes of the Software_Files Common Class 29
2-13 Attributes of the File Class 31
2-14 Attributes of the Control File Class 33
3-1 Software_spec Version Identifiers 40
3-2 Script Return Codes 57
3-3 Event Status 58
3-4 General Error Events 59
3-5 Session Events 61
3-6 Analysis Phase Events 63
3-7 Execution Phase Events 67
3-8 Return Codes 75
4-1 Default Levels 105
5-1 Example of Software Packaging Layout 126
5-2 File Attributes for INFO File 141
B-1 Possible Attributes of a Host Class 175

Preface

The Open Group

The Open Group is the leading vendor-neutral, international consortium for buyers and suppliers of technology. Its mission is to cause the development of a viable global information infrastructure that is ubiquitous, trusted, reliable, and as easy-to-use as the telephone. The essential functionality embedded in this infrastructure is what we term the *IT DialTone*. The Open Group creates an environment where all elements involved in technology development can cooperate to deliver less costly and more flexible IT solutions.

Formed in 1996 by the merger of the X/Open Company Ltd. (founded in 1984) and the Open Software Foundation (founded in 1988), The Open Group is supported by most of the world's largest user organizations, information systems vendors, and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to meet its new mission, as well as to assist user organizations, vendors, and suppliers in the development and implementation of products supporting the adoption and proliferation of systems which conform to standard specifications.

With more than 200 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- consolidating, prioritizing, and communicating customer requirements to vendors
- conducting research and development with industry, academia, and government agencies to deliver innovation and economy through projects associated with its Research Institute
- managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- adopting, integrating, and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- licensing and promoting the Open Brand, represented by the ‘‘X’’ mark, that designates vendor products which conform to Open Group Product Standards
- promoting the benefits of the IT DialTone to customers, vendors, and the public.

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development, and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trademark on behalf of the industry.

The Development of Product Standards

This process includes the identification of requirements for open systems and, now, the IT DialTone, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product.

The ‘‘X’’ device is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the Open Brand they guarantee, through the X/Open Trade Mark Licence Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical documentation, the main part of which is focused on specification development and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys, and business titles.

There are several types of specification:

- *CAE Specifications*

  CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our Product Standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. Where appropriate, they can demonstrate product compliance through the Open Brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *Preliminary Specifications*

  Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

  Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organizations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

  The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE,

OSF/Motif, and CDE.

Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

- *Product Documentation*

  This includes product documentation—programmer's guides, user manuals, and so on—relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

  These provide information that is useful in the evaluation, procurement, development, or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

  Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Program. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.

- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.opengroup.org/corrigenda**.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.opengroup.org/pubs**.

This Document

System administration utilities vary widely between vendors, and system administration is an area where to date no formal standards have achieved significant industry-wide acceptance.

This **Distributed Software Administration (XDSA)** specification, which is based on the **IEEE 1387.2 Software Administration Standard**, addresses this problem, for software administration in both stand-alone and distributed environments. XDSA defines a software packaging layout, a set of information maintained about software, and a set of utility programs to manipulate that software and information. It extends the IEEE 1387.2 Standard by adding significant functionality to deliver enhanced update and patch facilities.

This specification, like the **IEEE 1387.2 Standard**, specifies distributed operations without specifying the mechanism for how it is to be achieved. The Open Group has published a specification defining interoperability for XDSA, which uses the Distributed Computing Environment (DCE) Remote Procedure Call (RPC) mechanism — see referenced specification **XDSA-DCE**. The Open Group wishes to embrace other interoperability mechanisms for distributed XDSA working, so hopes to publish further such specifications as and when sufficient industry support for them becomes evident.

Structure

- **Chapter 1: General** — describes the scope, objectives, dependencies, and conformance issues related to this specification.
- **Chapter 2: Software Structures** — describes the software classes and attributes applicable to distributed software administration.
- **Chapter 3: Common Definitions for Utilities** — defines the common parts of the Systems Administration utilities which are defined in Chapter 4.
- **Chapter 4: System Administration Utilities** — defines each of the utilities which must be provided in a conformant implementation of this specification.
- **Chapter 5: Software Packaging Layout** — describes the components which make up the package: the directory structure, the software definition file formats, and the serial format of the layout in the directory structure.
- **Appendix A: Sample Files** — these are offered for information and guidance of implementors.
- **Appendix B: Rationale and Notes**— this gives background information on particular issues which have been recorded during development of the POSIX 1387.2 Standard and of this XDSA specification. It includes explanations and rationale which is considered likely to be useful to implementors and application writers.

A Glossary and Index are also provided.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for filenames, keywords, type names and environmental variables
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — attributes, data types and variable names

  - parameters (also called metavariables)
  - option arguments and extended options
  - command names
  - utilities; these are shown as follows: *name*
- Normal font is used for the names of constants and literals.
- Syntax, code fragments, and fixed values (for example, `true`) are shown in `fixed width` font.

Trademarks

Motif,® OSF/1,® UNIX,® and the ''X Device''® are registered trademarks and IT DialTone™ and The Open Group™ are trademarks of The Open Group in the U.S. and other countries.

Acknowledgements

The Open Group acknowledges the Institution of Electrical and Electronics Engineers, Inc., and the use of the **IEEE 1387.2 Standard** as the basis for development of this **Distributed Software Administration (XDSA)** specification.

Referenced Documents

The following documents are referenced in this specification:

ISO/IEC 646
: ISO/IEC 646: 1991, Information Processing — ISO 7-bit Coded Character Set for Information Interchange.

ISO POSIX-1
: ISO/IEC 9945-1: 1996, Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language] (identical to ANSI/IEEE Std 1003.1-1996). Incorporating ANSI/IEEE Stds 1003.1-1990, 1003.1b-1993, 1003.1c-1995 and 1003.1i-1995. ANSI/IEEE Standard 1003.1-1996[1].

ISO POSIX-2
: ISO/IEC 9945-2: 1993, Information Technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities (identical to IEEE Std 1003.2-1992 as amended by IEEE Std 1003.2a-1992).

ISO/IEC 10646
: ISO/IEC 10646-1: 1993, Information Technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.

POSIX 1387.2
: IEEE Std. 1387.2-1995, Information Technology — Portable Operating System Interface (POSIX) System Administration — Part 2: Software Administration.

XDSA-DCE
: CAE Specification, February 1997, Systems Management; Distributed Software Administration — DCE-RPC Interoperability (XDSA-DCE) (ISBN: 1-85912-137-3, C430), published by The Open Group.

The following documents provide additional bibliographical references for associated information:

[B1] Desktop Management Task Force, Desktop Management Interface Specification, Version 1.0, 29 April 1994.[2]

[B2] ISO 639: 1988, Code for the representation of names of languages.[3]

[B3] ISO/IEC 2022: 1994, Information processing — Character code structure and extension techniques.

[B4] ISO 2047: 1975, Information processing — Graphical representations for the control characters of the 7-bit coded character set.

---

1. IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, PO Box 1331, Piscataway, NJ 08855-1331, USA.
2. DMTF documents can be obtained via the World Wide Web from `http://www.dmtf.org/`
3. ISO/IEC documents can be obtained from the ISO Central Secretariat, Case postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse.

[B5] ISO 3166: 1993, Codes for the representation of names of countries.

[B6] ISO 4217: 1995, Codes for the representation of currencies and funds.

[B7] ISO/IEC 4873: 1991, Information technology — ISO 8-bit code for information interchange — Structure and rules for implementation.

[B8] ISO/IEC 6429: 1992, Information technology — Control functions for coded character sets.

[B9] ISO/IEC 6937: 1994, Information technology — Coded graphic character set for text communication — Latin alphabet.

[B10] ISO 8601: 1988, Data elements and interchange formats — Information interchange — Representation of dates and times.

[B11] ISO/IEC 8806: 1991, Information Technology — Computer graphics — Graphical Kernel System for Three Dimensions (GKS-3D) language bindings — Part 4: C.

[B12] ISO 8859, Information processing — 8-bit single-byte coded graphic character sets.

[B13] ISO/IEC 9899: 1990, Programming languages — C.[4]

[B14] ISO/IEC 10164-18: 1997: Information Technology — Open Systems Interconnection — Systems Management — Part 18: Software Management Function.

[B15] ISO/IEC 10646-1: 1993, Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.

[B16] ISO/IEC TR 10000-1: 1992, Information technology — Framework and taxonomy of International Standardized Profiles — Part 1: General principles and documentation framework.

[B17] ISO/IEC JTC 1 N1335, Final Report of ISO/IEC JTC 1 TSG-1 on Standards necessary to define Interfaces for Application Portability (IAP).

[B18] International Organization for Standardization/Association Française de Normalisation (ISO/AFNOR, 1989): Dictionary of Computer Science/Dictionnaire de L'Informatique.

[B19] IEEE Std 100-1992, IEEE Standard Dictionary of Electrical and Electronics Terms.

[B20] IEEE Std 1003.0-1995, IEEE Guide to the POSIX® Open Systems Environment (OSE).

[B21] IEEE P1003.1a/D12, Draft Revision to Information technology — Portable Operating System Interface (POSIX®) Part 1: System Application Program Interface (API) [C Language][5]

[B22] IEEE P2003/D7, Standard for Information Technology — &Test Methods for Measuring Conformance to POSIX®.

[B23] IEEE P2003.2/D11, Standard for Information Technology — Test Methods for Measuring Conformance to POSIX® — Part 2: Shell and Utilities.

---

4. IEC documents can be obtained from the IEC office, 3 rue de Varembé, Case Postale 131, CH-1211, Genève 20, Switzerland/Suisse.

5. Numbers preceded by ''P'' are IEEE authorized standards projects that were not approved by the IEEE Standards Board at the time this publication went to press. For information about obtaining drafts, contact the IEEE.

[B24] RFC 819, Su, Z. and Postel, J. B. Domain naming convention for Internet user applications.[6]

[B25] RFC 822, Crocker, D. — Standard for the format of ARPA Internet text messages.

[B26] RFC 920, Postel, J. B. and Reynolds, J. K. Domain requirements.

[B27] RFC 921, Postel, J. B., Domain name system implementation schedule — revised.

[B28] RFC 1123, Braden, R. T., Requirements for Internet hosts — application and support.

[B29] RFC 1514, Grillo, P. and Waldbusser, S., Host Resources MIB

[B30] American Telephone and Telegraph Company, System V (five) Interface Definition (SVID), Issues 2 and 3.[7]

[B31] University of California at Berkeley — Computer Science Research Group, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version, April 1986.

[B32] Guide, May 1992, Systems Management: Identification of Management Services (XIMS), (S190), published by The Open Group.

[B33] Guide, Sept 1993, Systems Management: Managed Object Guide (XMOG), (G302), published by The Open Group.

[B34] CAE Specification, March 1994, Systems Management: Management Protocols (XMP) API, (C306), published by The Open Group.

[B35] Guide, Sept 1993, Systems Management: Reference Model (XRM), (G207), published by The Open Group.

[B36] Preliminary Specification, August 1994, The Common Object Request Broker: Architecture and Specification, (C432), published by The Open Group and OMG[8].

[B37] XPG3, February 1992, Portability Guide, Issue 3 — 7-volume set plus Overview, (T010), published by The Open Group.

[B38] July 1996, The Single UNIX Specification — 5-volume set for UNIX 95, (T910), published by The Open Group.

---

6. Internet Requests for Comments (RFC) are available from the DDN Network Information Center, SRI International, Menlo Park, CA, USA 94025.

7. Available from AT&T, Morristown, NJ: UNIX Press, 1986, 1989.
This is one of several documents that represent an industry specification in a related area. The creators of such documents may be able to identify newer versions of relevance.

7. Available from The Regents of the University of California, Berkeley, CA, USA.

8. Joint publication with the Object Management Group.

Chapter 1

Introduction

1.1 Scope

This Software Administration specification defines a software packaging layout, and utilities that operate on that packaging layout as well as software installed from that packaging layout. The scope of this Software Administration specification is administration of software across distributed systems. This administration includes, but is not limited to, packaging of software for distribution, distribution of software to systems, installation and configuration of software on systems, removal of software from systems, and delivery of *updates* and *patches* packaged in 1387.2 format.

This Software Administration specification is motivated by many factors, including a desire by system administrators and software suppliers to have a common way of installing and removing software. To meet the needs of these groups, this Software Administration specification consists of several components, listed below. The readers of this Software Administration specification include system administrators, suppliers of software that implement this Software Administration specification, and suppliers of software that use implementations of this Software Administration specification. Readers in each of these categories may find their attention drawn to different sections.

The key components are listed below.

Software structures
: This Software Administration specification defines a hierarchical set of structures used to define software. Information is kept about the software based on these structure definitions. The structure definitions apply both to installed software and to software prepared for installation but not yet installed.

Software packaging layout
: This Software Administration specification defines the organization of software on a distribution medium, the information held about that software, and the way in which such information is represented. This enables both portability of software distributions across systems of different architecture, and the use of different media to distribute software (including both file system and serial image forms).

Information kept about software
: This Software Administration specification defines the information that is held about software, both installed software and distributions. This definition provides a consistent view of software, even when that software is provided from various sources. The way in which the information is held is undefined within this Software Administration specification.

Utilities to administer software
: This Software Administration specification defines a utility to convert software into the packaging layout, known as a distribution. This Software Administration specification also contains utilities to examine the information in a distribution, copy software from one distribution to another, install software from a distribution, remove software from a distribution, and verify the integrity of a distribution. There are also utilities for configuring installed software, patching software, verifying the integrity of installed software, examining and modifying the information held about installed software, and for removing

installed software from a system. This provides administrators a consistent method of dealing with software across all conforming systems.

Distributed software administration
This Software Administration specification defines the concepts, and the utility syntax and behaviors, for managing software in a distributed environment. This includes the concept of different software administration roles (developer, packager, manager, source, target and client). Different utilities involve different roles, and different roles may be distributed across multiple systems within a single command execution.

Although not requiring a fully conformant POSIX.1 base (ISO/IEC 9945-1 — see referenced documents) and POSIX.2 (ISO/IEC 9945-2 — see referenced documents), this Software Administration specification is based upon the knowledge of, and documentation for, existing programs that assume an interface and architecture similar to that described by POSIX.1 and POSIX.2. Any questions regarding the definition of terms or the semantics of an underlying concept should be referred to POSIX.1 and POSIX.2.

This Software Administration specification does not require the use of any specific programming language and, in particular, does not require the use of the C language. It is based upon the knowledge of, and documentation for, existing programs that utilize C-language interfaces. Any questions regarding the definition of terms or the semantics of an underlying concept in this language should be referred to the **C Standard** (ISO/IEC 9989 — see referenced documents).

1.2 Dependencies

1.2.1 Features Inherited From POSIX.1

This section describes some of the features provided by **POSIX.1** (ISO/IEC 9945-1 — see referenced documents) which are assumed to be globally available to all conforming implementations. This section does not attempt to detail all the POSIX.1 features that are required by all the utilities defined in this Software Administration specification. The utility descriptions point out additional functionality required to provide the corresponding features needed.

The following description explains frequently used concepts. Utility description statements override these defaults when appropriate.

File System

The hierarchical directory structure of POSIX.1 is assumed to be available, as well as support for case-sensitive file names. In addition, various file attributes are also assumed to be present, including the following: *type, owner, group, mode, uid, gid, mtime, major, minor*.

Environment Variables

The existence of environment variables in general is assumed, as well as **PATH**, **LANG**, **LC_ALL**, **LC_CTYPE**, **LC_MESSAGES**, **LC_TIME**, **TZ**.

Data Interchange Format

The ability to read and write the data interchange formats of POSIX.1 is assumed, including both extended **tar** and extended **cpio**. See POSIX.1. See also this Specification Section 5.3 on page 147.

1.2.2 Features Inherited From POSIX.2

This section describes some of the features provided by (POSIX.2) which are assumed to be globally available to all systems conforming to this specification. It does not attempt to detail all of the POSIX.2 features that are required by all the utilities and control scripts defined in this Software Administration specification. Additional functionality required may be found in the utility descriptions and in Section 3.6.1 on page 56.

All of the utilities defined in POSIX.2 are required, including the shell interpreter (`sh`). This assures a portable environment for executable control files.

1.3 Conformance

1.3.1 Full Conformance

A conforming implementation must support all interfaces defined within this Software Administration specification. These interfaces must support all the functional behavior described herein. The interfaces covered by this definition of conformance include, but are not limited to, utilities and their options and extended options, the behavior of the utilities, including the generation of events, structures, attributes and their values, and file formats.

The system may provide additional or enhanced utilities, functions, or facilities not required by this Software Administration specification. Optional extensions should be identified explicitly. Optional extensions should conform to ISO/IEC 9945-2 — see referenced documents, section 2.10.2 (utility syntax guidelines"). Optional extensions, when used, may change the behavior of utilities, functions, or facilities defined by this Software Administration specification. In such cases, the conformance statement for the implementation must define an execution environment (that is, general operating instructions) in which a conformant implementation may be operated upon and yield the behavior specified by this Software Administration specification. In no case must such an environment require modification of a conformant implementation.

1.3.2 Limited Conformance

A limited conformance implementation must meet all of the criteria established for a fully conformant implementation, with the following exception:

- For the value of `HOST` in specifications of sources and targets (see Section 3.4.2 on page 42, the system may support only the local machine. While this type of limited conformance removes support for remote operations, the syntax of all utilities and files must remain identical to that required for fully conformant implementations. The way in which this limitation is imposed by the implementation must be implementation defined.

Chapter 2

Software Structures

2.1 Classes and Attributes

This section describes the software classes and attributes applicable to software administration. Each utility in section Chapter 4 describes the operations on the software objects including how the values of the attributes affect the behavior of the operations. Whether these operations and behaviors are implemented as procedures on software structures or by other means is undefined within this Software Administration specification.

The software administration classes form a hierarchy that consists of: distributions, media, installed_software, categories, vendors, bundles, products, subproducts, filesets, control_files, and files.

At each level, this hierarchy is defined by containment attributes which reference objects at lower levels. Operations on objects of lower levels, such as files, are actually enacted by operations on objects of higher levels. For example, files may be created in a distribution by copying a software product.

A ‘‘common class’’ is used to define attributes that are common between related objects. Objects inherit attribute definitions from common classes as well as their individual attributes. This provides logical relationship between the objects that share the same common class. The software administration common classes are: software_collection, software, and software_file.

Objects that share the same common class are also referred to generically as: software_collections, software objects, and software_files.

In tables in the following sections, attributes are listed with various properties. The attributes and their values manifest themselves as part of the utilities defined in section Chapter 4 and the software packaging layout in section Chapter 5.

The names of attributes are as provided. If the underlying host allows for the distinction of case, the attribute names is sensitive to case. Where values of attributes are shown, if the underlying host allows for the distinction of case, the values of attributes is sensitive to case. If the underlying host does not allow for the distinction of case for either the name or value of an attribute, the way in which case differences are handled is implementation defined.

The attribute tables in subsequent sections list the following information:

Attribute
: The name of the attribute, also used as the keyword for the attribute.

Length
: The maximum permitted length of the value of the attribute.

All attribute values in this Software Administration specification are represented only as strings. The length is the maximum permitted length of the value in bytes or, for attributes whose values are lists, the maximum permitted number of items permitted in the list. Since the means of storing such data for installed software is undefined within this Software Administration specification, an implementation may store such values internally in different structures for installed software. See the descriptions for ***distribution catalog***, ***exported catalog*** and ***installed_software catalog*** in the Glossary.

Permitted Values

The character sequences permitted as values for this attribute.

Default Value

The value of the attribute if the attribute is not specified.

A default value of `None` means the system will not supply a value in cases where the attribute has not been specified and the attribute is not one whose values are generated dynamically. See Section 5.1.1 on page 126 and Section 5.2 on page 130.

The attribute tables are broken into three groups:

- The top group contains the attributes that are used to identify a particular instance.
- The middle group contains the rest of the attributes that describe other information or behaviors for the object.
- The bottom group contains the attributes that describe the objects contained within this object. The way in which these lists are represented in software definition files is described in Section 5.2 on page 130. The way in which these lists are represented by `swlist` is described in *swlist* on page 104.

Beyond this convention, the order of attributes shown in this section is not significant. For any attribute ordering rules, see Section 5.2 on page 130. Some attributes do not apply to software objects in both distributions and installed_software objects. See Section 5.2 on page 130 for details.

Management of lists of software_collections contained within a host is undefined within this Software Administration specification. See Section 3.4.2 on page 42 for the way in which software_collections are identified relative to a software host.

2.2 Software_Collection

A software_collection is the common class from which distribution and installed_software objects inherit.

A software_collection can contain product and bundle software objects. A Software_collection can contain multiple versions of the same product or bundle software objects, namely products or bundles that share the same value for the *tag* attribute.

Each software_collection has a catalog associated with it that contains the metadata describing all software objects in that collection.[1]

Table 2-1 Attributes of the Software_Collection Common Class

Attribute	Length	Permitted Values	Default Value
path	Undefined	Pathname character string	Implementation defined
dfiles	64	Filename character string	`dfiles`
layout_version	64	`1.0`	`1.0`
pfiles	64	Filename character string	`pfiles`
bundles	Undefined	List of `bundle_software_specs`	Empty list
products	Undefined	List of `product_software_specs`	Empty list

Software_Collection Attributes

The following attributes describe each instance of the software_collection class, and are inherited by each instance of the distribution and installed_software classes:

bundles
A list of `bundle_software_specs`.

Each `software_spec` refers to a bundle. Each `software_spec` is fully qualified. See Section 3.4.1 on page 38 for the syntax of `software_spec`.

dfiles
The name of the directory in the exported catalog structure below which any attributes stored as files for the software_collection are stored (see Section 5.1 on page 126.

layout_version
This attribute, and its value, are included for future use.

path
The identifier for a particular software collection on a host.

1. For distribution software_collections, the catalog information is stored in the software packaging layout in an exported catalog structure. For installed_software objects, how the catalog information is stored (whether in a file or database, for example) is undefined within this Software Administration specification.

The value of the path attribute is an absolute path. The default value of this attribute is implementation defined. See Section 3.5.2 on page 44.

pfiles

The name of the directory in the exported catalog structure below which any control_files, and attributes stored as files, for the product are stored (see Section 5.1 on page 126.

products

A list of `product_software_specs`.

Each `software_spec` refers to a product. Each `software_spec` is fully qualified. See Section 3.4.1 on page 38 for the syntax of `software_spec`.

2.3 Distribution

A distribution contains product and bundle software objects. It is contained on a distribution media or may be part of the file store of a system. The distribution may contain a variety of software products and bundles, and that software may be applicable to a variety of hardware architectures or operating systems.

The distribution class inherits attributes from the software_collection common class.

A particular distribution object is identified within a host by the *path* attribute. For distributions, the *path* attribute is the pathname to the directory containing a distribution in the directory format of the software packaging layout, or a file or device file containing a distribution in a serial format of the software packaging layout.

Distributions can contain more than one version of a product or bundle. A version is uniquely identified within a distribution by the values of the *revision*, *vendor_tag*, and *architecture* attributes.

Table 2-2 Attributes of the Distribution Class

Attribute	Length	Permitted Values	Default Value
uuid	64	Portable character string	Empty string
media	Undefined	List of media *sequence_number* values	Empty list

Distribution Attributes

These attributes, along with the attributes listed in Table 2-1 on page 7, describe each instance of the distribution class:

media

A list of *media.sequence_number* values for the distribution if the distribution spans multiple media. Each medium in a distribution has its *media.sequence_number* in the `INDEX` file defined for that medium. See Section 5.3 on page 147. An implementation may include definitions for all media in the global `INDEX` file found on the first medium in the distribution. The *media.sequence_number* for the first medium in the distribution is `1` and is the first item in the list.

uuid

A string that should uniquely identify a distribution.

The way in which a unique string is generated is undefined. This attribute is used for determining whether subsequent media are from the same set as the one that an install or copy started with. This attribute is defined for distributions that span multiple media.

2.4 Media

The media class is used to describe the media attributes for distributions that span multiple media.

Table 2-3 Attributes of the Media Class

Attribute	Length	Permitted Values	Default Value
sequence_number	64	Portable character string	`1`

Media Attributes

sequence_number

Identifies a particular media when a distribution spans multiple media.

It is used for identifying the correct medium on which to find the distribution files when the distribution spans multiple media.

2.5 Installed_Software

The installed_software class is used to describe the bundle and product software that has been installed on a file system.

The installed_software class inherits attributes from the software_collection common class.

A particular installed_software object is identified within a host by both the *path* attribute (defined in the software_collection class) and the *catalog* attribute. For installed_software objects, the *path* attribute is the root directory for the installed_software object below which all the software files were installed.

An installed_software object can contain multiple versions of a product or bundle. Multiple product and bundle versions are distinguished by the same attributes as distribution products, plus the user-specifiable *location* and *qualifier* attributes. Multiple product versions may be installed at the same time in an installed_software object. Different product versions may be installed into different locations, and different filesets from different product versions may be installed in the same location.

Table 2-4 Attributes of the Installed Software Class

Attribute	Length	Permitted Values	Default Value
catalog	Undefined	Portable character string	Undefined

Installed_Software Attributes

This attribute, along with the attributes listed in Table 2-1 on page 7, describe each instance of the installed_software class:

catalog

Along with the *path* attribute, identifies a single installed_software object.

Different installed_software objects may have the same value for the *path* attribute if and only if the value of their *catalog* attributes are different.

The *catalog* attribute is evaluated relative to the *path* attribute. It may be a POSIX.1 pathname or other identifier: together they form the key to the undefined catalog storage for this installed_software object.

2.6 Vendor

The vendor class is used to describe the attributes of the vendors associated with products and bundles.

Each product or bundle identifies a vendor with a vendor_tag that identifies a particular vendor object. The *vendor_tag* attribute is used to distinguish products and bundles from different vendors that share the same product or bundle *tag*.

Table 2-5 Attributes of the Vendor Class

Attribute	Length	Permitted Values	Default Value
tag	64	Filename character string	Empty string
title	256	Portable character string	Empty string
description	Undefined	Portable character string	Empty string

Vendor Attributes

description
A more detailed description of the vendor or information about the vendor.

tag
A short identifying name of the vendor that supplied the product.

This attribute is used to to distinguish products and bundles from different vendors, and for resolving software specifications. Each software vendor should attempt to have a unique value for the *tag* attribute.

title
A longer name of the vendor that supplied the product. It is used for presentation purposes.

2.7 Category

Category Attributes

The *category attributes* class is used to describe the attributes of the category attributes associated with products and bundles.

Each product or bundle identifies a category attribute that identifies a particular object. *tag*.

Table 2-6 Attributes of the Category Class

Attribute	Length	Permitted Values	Default Value
tag	64	Filename character string	None
title	256	Portable character string	Empty string
description	Undefined	Portable character string	Empty string
revision	64	Portable character string	Empty string

Category Attributes

tag
A short name identifying the category. Each category must have a unique tag. This attribute is used to identify a particular category object that a software item identifies through one of its attributes. The tag *patch* is reserved as a built-in category for filesets with the *is_patch* attribute.

title
A longer name of the category used for presentation purposes.

description
A more detailed description of the category.

revision
This attribute is only used to determine which category object definition to maintain in a software collection when one being installed or copied does not match the one already in the software collection for that *category.tag*. The category definition with the higher revision is maintained.

2.8 Software

Software is the common class from which products, bundles, filesets and subproducts inherit.

Table 2-7 Attributes of the Software Common Class

Attribute	Length	Permitted Values	Default Value
tag	64	Filename character string	None
category_tag	Undefined	list of *category.tag* values	Empty list, or *patch* if the object has the *is_patch* attribute set to `true`
create_time	16	Integer character string	None
description	Undefined	Portable character string	Empty string
is_patch	8	One of: `true, false`	`false`
mod_time	16	Integer character string	None
size	32	Integer character string	None
title	256	Portable character string	Empty string

Software Common Attributes

The following attributes describe each instance of the software common class, and are inherited by each instance of the product, bundle, fileset, and subproduct classes:

category_tag

A repeatable tag based attribute identifying a set of categories that a software object is a member of. This is used as a selection mechanism, and can be used independent of patches.

Like *vendor_tag*, this attribute optionally is also a pointer to a *category* object that contains additional information about this category (a ‘‘title’’ one-line definition, and a ‘‘description’’ of the category).

All software items with the attribute of *is_patch* set to `true`, have a built-in category of *patch* automatically included. A category of *patch* can not be specified in the PSF file.

create_time

A value which is set by the implementation to be the time that the catalog information for this object was first written.

Time is represented as seconds since the Epoch, as defined in POSIX.2.

description

A more detailed description of the software object.

is_patch

A new Boolean attribute, *is_patch*, will be defined to indicate that a software object is to be identified as a patch.

Only filesets with the is_patch attribute have patched files in this proposal. The other levels can be identified as patches for the listing utilities to facilitate identification of patch software at any level.

Patch filesets have particular behaviors:

- Patch filesets are in general similar in operation to normal filesets except that they have an explicit or implicit ancestor with which they merge during installation, have the ability to be rolled back, and maintain catalog information to support these features.
- A patch fileset can be installed in the same session as its base, or ancestor, fileset. Patch filesets will always be installed after the base fileset if installed in the same session.
- Control scripts delivered with the patch fileset will only run when that patch fileset is installed. They do not replace the control scripts for the base fileset.

mod_time

A value which is set by the implementation to be the time that the catalog information for this object was last written.

Time is represented as seconds since the Epoch, as defined in POSIX.1.

size

The sum of the sizes in bytes of all files and control_files contained within the software object.

For objects other than filesets, the value is computed dynamically as required. See Section 5.2.7 on page 137, Section 5.2.8 on page 138, and Section 5.2.9 on page 139.

tag

A short name associated with the software object.

It is the one attribute that is always required to identify a software object. For more information on software selections, see Section 3.4.1 on page 38.

title

A longer name associated with the software object, used for display purposes.

2.9 Products

Products can contain filesets, which can be grouped into subproducts. Products are named by their *tag* attributes. A particular product object is uniquely identified within a software_collection by the *tag* attribute and by the version distinguishing attributes. The attributes that uniquely distinguish a particular product version within a software_collection are ***revision***, ***architecture***, ***vendor_tag***, ***location***, and ***qualifier***.

The product class inherits the attributes of the software common class.

See subsection Software Compatibility within Section 3.4.1 on page 38.

Table 2-8 Attributes of the Product Class

Attribute	Length	Permitted Values	Default Value
architecture	64	Portable character string	Empty string
location	Undefined	Pathname character string	<*product.directory*>
qualifier	64	Portable character string	Empty string
revision	64	Portable character string	Empty string
vendor_tag	64	Filename character string	Empty string
all_filesets	Undefined	List of fileset *tag* values	Empty list
control_directory	Undefined	Filename character string	<*product.tag*>
copyright	Undefined	Portable character string	Empty string
directory	Undefined	Pathname character string	`/`
instance_id	16	Filename character string	`1`
is_locatable	8	One of: `true`, `false`	`true`
postkernel	Undefined	Pathname character string	Implementation defined
layout_version	64	`1.0`	`1.0`
machine_type	64	Software pattern matching string	Empty string
number	64	Portable character string	Empty string
os_name	64	Software pattern matching string	Empty string
os_release	64	Software pattern matching string	Empty string
os_version	64	Software pattern matching string	Empty string
control_files	Undefined	List of *control_file*tag values	Empty list
subproducts	Undefined	List of *subproduct*tag values	Empty list
filesets	Undefined	List of *fileset*tag values	Empty list

Product Attributes

The product attributes, along with the attributes listed in Table 2-7 on page 14, describe each instance of the product class:

all_filesets
This is a list of all filesets defined for the product, as opposed to what is currently installed, described by the *filesets* attribute. The *all_filesets* attribute is used to determine completeness of this product when another software object has a dependency on this product. In checking a product prerequisite or corequisite, the existence of a *fileset*tag in *all_filesets* that is not actually `installed` or `available` indicates that the dependency is not satisfied.

This does not affect exrequisites as they test whether any of the contents of the dependency specification are present instead of all of the contents tested for prerequisites or corequisites.

architecture
A vendor-defined string used to distinguish variations of a product.

It is used for presentation purposes and for resolving software specifications. If a product with the same value of the *revision* and *vendor_tag* attributes has different versions of software for different target architectures, or any other variation (such as supported locale), then the value of the *architecture* attribute is different for each version. No additional semantics is assumed for its value.

control_directory
The name of the product control directory below which the control_files for the product are stored within an exported catalog.

See Section 5.1.

control_files
A list of the values of the *tag* attribute for all the control_files in the product.

These scripts are executed before and after software load, and before and after software removal.

copyright
The copyright notice for the product.

directory
The vendor-defined directory commonly associated with the product.

Generally, this will be the directory in or below which all (or mostly all) files within the product are installed.

For a product which has filesets with *is_locatable* equals `true`, all files which contain this directory as the first part of their path can be relocated to the *location* directory during installation by replacing the *product*directory portion with the *product*location.

filesets
A list of the values of the *tag* attribute for all the filesets in the product which are currently `installed` (in an installed_software object) or `available` (in a distribution).

instance_id
A single attribute that distinguishes versions of products (and bundles) with the same tag.

It is a simple form of the version distinguishing attributes, valid only within the context of an exported catalog.

is_locatable
A boolean value indicating whether any of the filesets in the product have the *is_locatable*

attribute set to `true`.

layout_version
This attribute, and its value, are included for future use.

location
Used for resolving `software_specs` for installed software.

A specific product location refers to all filesets of that product that are installed at that location. This is the path beneath which the relocatable files of that product are stored. See subsection ‘‘File Location’’ in *swinstall* on page 94.

This attribute is valid only for products in installed_software.

machine_type
A software pattern matching string describing valid machine members of the *uname* structure as defined by POSIX.2.

It is used for determining compatibility.

number
The semantics associated with the values of this attribute are undefined.

This attribute can be used to store such vendor-defined values as part number, order number or serial number.

os_name
A software pattern matching string describing valid sysname members of the *uname* structure as defined by POSIX.2.

It is used for determining compatibility.

os_release
A software pattern matching string describing valid release members of the *uname* structure as defined by POSIX.2.

It is used for determining compatibility.

os_version
A software pattern matching string describing valid version members of the *uname* structure as defined by POSIX.2.

It is used for determining compatibility.

postkernel
The path to the script that is run after the kernel filesets have been installed.

Any product containing kernel filesets should include this path. If this attribute is supplied, the corresponding script is executed used if it exists relative to the root directory of the installed_software. If this attribute is not supplied, then the implementation defined path (the default value for the attribute) is used if it exists relative to the root directory of the installed_software. Note that the use of an alternate root directory may mean that the default path does not exist relative to the root directory of the installed_software .

qualifier
Specified by a user when installing software and used for identifying a product (or set of product versions) using a logical name.

Applies only to products in installed_software.

revision
A vendor-defined string describing the revision of the product.

It is used for presentation purposes and for resolving software specifications. The revision is interpreted as a . (period) separated string. See Section 3.4.1 on page 38.

subproducts
A list of the values of the *tag* attribute for all the subproducts in the product.

vendor_tag
A short identifying name of the vendor that supplied the product.

This attribute may also be used to identify a vendor object containing additional attributes describing the vendor.

This attribute is used to distinguish software objects, allowing more than one vendor to produce a product with the same value of the other version distinguishing attributes. It is used for presentation purposes and for resolving software specifications.

2.10 Bundles

Bundles are groupings of software objects. Bundles contain references to products, parts of products, or other bundles. A software object can be referenced by more than one bundle.

The bundle class inherits the attributes of the software common class.

A particular bundle object is uniquely identified within a software_collection by the tag and by the version distinguishing attributes. The attributes that uniquely distinguish a particular bundle version are *revision*, *architecture*, *location*, *vendor_tag*, and *qualifier*.

Bundles, like products, are named by their *tag* attributes and share the same name space as products. Products and bundles is considered together in determining a unique value for *instance_id*.

Bundles and products include many of the same attributes. No bundle attributes are automatically derived from the contained product attributes. They are defined independently. See subsection ‘‘Software Compatibility’’ in Section 3.4.2 on page 42.

Bundle definitions are copied or installed when explicitly specified in a software selection for `swcopy` and `swinstall` respectively. They remain installed until explicitly removed or until all of their contents are removed.

Table 2-9 Attributes of the Bundle Class

Attribute	Length	Permitted Values	Default Value
architecture	64	Portable character string	Empty string
location	Undefined	Pathname character string	<<*bundle.directory*>
qualifier	64	Portable character string	Empty String
revision	64	Portable character string	Empty string
vendor_tag	64	Filename character string	Empty string
contents	Undefined	List of `software_specs`	Empty list
copyright	Undefined	Portable character string	Empty string
directory	Undefined	Pathname character string	Empty string
instance_id	16	Filename character string	`1`
is_locatable	8	One of: `true`, `false`	`true`
layout_version	64	`1.0`	`1.0`
machine_type	64	Software pattern matching string	Empty string
number	64	Portable character string	Empty string
os_name	64	Software pattern matching string	Empty string
os_release	64	Software pattern matching string	Empty string
os_version	64	Software pattern matching string	Empty string

Bundle Attributes

These attributes, along with the attributes listed in Table 2-7 on page 14, describe each instance of the bundle class:

architecture
A vendor-defined string used to distinguish variations of a bundle.

It is used for presentation purposes and for resolving software specifications.

contents
A list of `software_specs` that defines the list of software grouped into this bundle, as originally defined in the PSF.

copyright
A copyright notice for the bundle.

directory
The default directory (and location) of the bundle.

This is the default path prepended, when the bundle is installed, to the location of each product and bundle specification within this bundle.

instance_id
A single attribute that distinguishes versions of bundles (and products) with the same tag.

It is a simple form of the version distinguishing attributes, valid only within the context of an exported catalog.

is_locatable
A boolean value indicating whether any of the contents in the bundle have the *is_locatable* attribute set to `true`.

layout_version
This attribute, and its value, are included for future use.

location
An attribute whose value is set when installing software and used for resolving `software_specs` for installed software.

When installing a bundle, the *bundle* location is prepended to the location specification for each `software_spec` in the contents of the bundle, before that `software_spec` is resolved.

The *contents* attribute of the bundle is not modified.

Applies only to bundles in installed_software.

machine_type
A software pattern matching string describing valid machine members of the *uname* structure as defined in POSIX.2.

It is used for determining compatibility.

number
The semantics associated with the values of this attribute are undefined.

This attribute can be used to store such vendor-defined values as part number, order number or serial number.

os_name
A software pattern matching string describing valid sysname members of the *uname*

structure as defined in POSIX.2.

It is used for determining compatibility.

os_release
A software pattern matching string describing valid release members of the *uname* structure as defined in POSIX.2.

It is used for determining compatibility.

os_version
A software pattern matching string describing valid version members of the *uname* structure as defined in POSIX.2.

It is used for determining compatibility.

qualifier
Specified by a user when installing software, and used for identifying a bundle (or set of bundle versions) using a logical name.

Applies only to bundles in installed_software.

revision
A vendor-defined string used to distinguish different revisions of bundles from one another.

It is used for presentation purposes and for resolving software specifications.

vendor_tag
A short identifying name of the vendor that supplied the bundle.

This attribute is used to identify a vendor object containing additional attributes describing the vendor.

This attribute is used to distinguish bundles, allowing more than one vendor to produce a bundle with the same value of the *tag* attribute.

2.11 Filesets

The fileset class is used to define a set of software files. The fileset is the smallest level of software that can be managed by the tasks defined in this standard.

The fileset class inherits attributes from the software common class.

Filesets contain the actual files and control_files that make up the software product.

A particular fileset object is identified within a product by the *tag* attribute.[2]

---

2. A fileset is strictly contained within the product. There can not be more than one fileset in the product with the same tag. A fileset can not be in more than one product. However, a product may be referenced by more than one bundle.

Table 2-10 Attributes of the Fileset Class

Attribute	Length	Permitted Values	Default Value
ancestor	Undefined	list of fileset software_specs of the form *product.fileset,version*	*product.tag* *.fileset.tag* ,r<*revision* ,a=*architecture* ,v=*vendor_tag*
applied_patches	Undefined	List of patch software_specs of the form *product.fileset,version*	Empty list
control_directory	Undefined	Filename character string	<*fileset.tag*>
corequisites	Undefined	List of `dependency_specs`	Empty list
exrequisites	Undefined	List of `dependency_specs`	Empty list
is_kernel	8	One of: `true`, `false`	`false`
is_locatable	8	One of: `true`, `false`	`true`
is_reboot	8	One of: `true`, `false`	`false`
is_sparse	8	One of: `true`, `false`	`false`
location	Undefined	Pathname character string	<*product.directory*>
media_sequence_number	Undefined	List of *media*sequence_number values	`1`
patch_state	16	One of `applied`, `committed`, `superseded`	None
prerequisites	Undefined	List of `dependency_specs`	Empty list
revision	64	Filename character string	None
saved_files_directory	Undefined	Pathname character string	Empty string
state	16	One of: `configured`, `installed`, `corrupt`, `removed`, `available`, `transient`	None
supersedes (patch)	Undefined	list of patch software_specs	Empty list
supersedes (update)	Undefined	list of fully qualified fileset software_specs of the form *product.fileset,version*	Empty list
superseded_by	Undefined	patch software_spec of the form *product.fileset,version*	Empty string
control_files	Undefined	List of *control_file*tag values	Empty list
files	Undefined	List of *file*path values	Empty list

Fileset Attributes

These attributes, along with the attributes listed in Table 2-7 on page 14, describe each instance of the fileset class:

ancestor
: This attribute is used with a *patch_match_target* or *update_match_target* option. It designates an ancestor fileset to check for when the *match_target* option is set to `true`. The default for this option, if not defined, is any fileset with the same tags and other version distinguishing attributes, but with a lower product revision.

applied_patches
: The *applied_patches* attribute is used to determine the list of patches that have been applied to a base fileset. It can be thought of as the inverse of the *ancestor* attribute.

  If this attribute is an empty list, then this fileset has no patches applied to it.

control_directory
: The name of the fileset control directory below which the control_files for the fileset are stored within an exported catalog. See Section 5.1 on page 126.

control_files
: A list of the values of the *tag* attribute for the control_files in the fileset

corequisites
: A list of `dependency_specs` for software required to be installed and configured for this fileset to work.

  Dependencies is considered when copying, installing, configuring, verifying, and removing software. See *swconfig* on page 82 (*swconfig* analysis phase), *swcopy* on page 86 (*swcopy* analysis phase), *swinstall* on page 93 (*swinstall* analysis phase), *swremove* on page 116 (*swremove* analysis phase), and *swverify* on page 122 (*swverify* analysis phase).

  The software specified by the `dependency_spec` must be complete in order for the dependency to be resolved successfully. See *all_filesets* in Section 2.9.0 on page 17.

exrequisites
: A list of `dependency_specs` for software required not to be installed when this fileset is installed.

  Dependencies is considered when installing, configuring, verifying, and removing software. See *swconfig* on page 82 (*swconfig* analysis phase), *swinstall* on page 93 (*swinstall* analysis phase), *swremove* on page 116 (*swremove* analysis phase), and *swverify* on page 122 (*swverify* analysis phase).

  No part of the software specified by the `dependency_spec` may be installed in order for this dependency to be resolved successfully.

files
: A list of the values of the *path* attribute for the files in the fileset.

is_kernel
: A boolean value indicating the fileset requires a kernel rebuild.

is_locatable
: A boolean value indicating if the fileset may be re-located during installation.

is_reboot
: A boolean value indicating the host on which the fileset is configured should be re-booted.

is_sparse

This fileset denotes a fileset that is not complete, but one that has been qualified as an update (as opposed to a patch). One outcome of updating via a sparse fileset is that the catalog information from the old fileset is merged into the new fileset and the old fileset is then removed, leaving the system in the same state as it would be after an update of a full fileset.

This option should be used in conjunction with an *ancestor* attribute showing exactly which version or versions of software this sparse fileset can update. Filesets that are sparse are only useful when installed along with those versions, or when those versions are already installed.

location

Specifies the location below which relocatable files are stored.

This attribute is only valid for filesets in installed software. It differs from the *product* directory attribute only if relocation was specified during installation. See ''File Location'' subsection in *swinstall* on page 94.

media_sequence_number

Identifies the *media.sequence_number* for the medium on which the files for this fileset is found.

If a single fileset spans multiple media, this attribute identifies a list of *media.sequence_number* values, identifying all of the media on which the fileset is found. In that case, the order of the list is interpreted as the order in which to read the media. See Section 2.3 on page 9, Section 2.4 on page 10, and Section 5.3 on page 147.

patch_state

The *patch_state* attribute only applies to installed patches and characterizes the current state of an installed patch.

prerequisites

A list of `dependency_specs` for software required to be installed prior to the installation of this fileset and configured prior to the configuration of this fileset.

Dependencies is considered when copying, installing, configuring, verifying, and removing software. See *swconfig* on page 82 (*swconfig* analysis phase), *swcopy* on page 86 (*swcopy* analysis phase), *swinstall* on page 93 (*swinstall* analysis phase), *swremove* on page 116 (*swremove* analysis phase), and *swverify* on page 122 (*swverify* analysis phase).

The software specified by the `dependency_spec` must be complete in order for the dependency to be resolved successfully. See *all_filesets* in Section 2.9.0 on page 17

Circular definitions should be avoided within package definitions. Behavior when circular definitions are encountered is implementation defined.

revision

Defines the revision of the fileset.

It is used for presentation purposes and for resolving software specifications. A fileset revision, but with no behavioral value, must be specified if the patch has the same product version and same fileset tag as the fileset it patches. Two filesets with the same revision can now exist in the same product version — the base fileset and one or more patch filesets.

saved_files_directory

The value of the *saved_files_directory* that *swinstall* used to save the patched files if *patch_save_files* was set to `true`. When rolling back or committing this patch, this attribute is used to determine the directory to access the saved files.

The value of this option should be an absolute path. If the value is relative, the behavior is undefined. If the value is the empty string, then the behavior is undefined.

state

An indication of the current status of the fileset.

This attribute may have one of the following values: `configured`, `installed`, `corrupt`, `removed`, `available`, and `transient`.

supersedes (for patch)

The *supersedes* attribute is used when a patch is replaced by (or merged into) a later patch. The attribute indicates which previous patches are replaced by the patch being installed or copied. This attribute is repeatable in order to facilitate merging of patches.

This attribute is a list of software specifications of other patches that this patch supersedes. When a patch supersedes another patch, the superseding patch is the one that is automatically selected by default (similar to the 1387.2 selection hueristic of ‘‘highest compatible versions’’). See *swinstall* on page 92. A superseding patch will replace the files of the patch it supersedes when installed after that patch.

supersedes (for update)

This attributes designates that this fileset supersedes a previous fileset. The behavior associated with a fileset that supersedes another fileset is similar to the behavior of a fileset that is simply a later revision of the same fileset. In particular, the superseding fileset:

- Is chosen over the superseded fileset when selecting the *highest compatible version* of a fileset that was not fully specified
- Meets the dependencies of filesets depending on the superseded fileset
- Causes removal of the catalog information of the superseded fileset when it is installed into the same location

If multiple filesets are superseded by the same fileset, then this behavior applies to each of those filesets.

superseded_by

The *superseded_by* attribute lists what patch superseded this patch.

2.12 Subproducts

Subproducts are groupings of filesets and subproducts within a single product. Subproducts do not contain filesets or subproducts within the name space of the subproduct, but instead refer to to them. A subproduct can refer to another subproduct. A subproduct or fileset can be referenced by more than one subproduct.

The subproduct class inherits the attributes of the software common class.

A particular subproduct object is named, and identified within a product, by the *tag* attribute. The values of the *tag* attribute of all subproducts and filesets is unique within a product.

Subproduct definitions are copied or installed when any fileset specified in the contents of the subproduct is copied or installed with `swcopy` or `swinstall` respectively. They remain installed until explicitly removed or until all of their contents are removed.

Table 2-11 Attributes of the Subproduct Class

Attribute	Length	Permitted Values	Default Value
contents	Undefined	List of *tag* values	Empty list

Subproduct Attributes

These attributes, along with the attributes listed in Table 2-7 on page 14 describe each instance of the subproduct class:

contents
A list of *tag* values that defines the list of filesets and subproducts grouped into this subproduct.

2.13 Software_Files

Software_file is the common class that files and control_files inherit from. A software_file is a file as defined in POSIX.1.

Table 2-12 Attributes of the Software_Files Common Class

Attribute	Length	Permitted Values	Default Value
cksum	16	Integer character string	None
compressed_cksum	16	Integer character string	None
compressed_size	16	Integer character string	None
compression_state	16	One of: `uncompressed`, `compressed`, `not_compressible`	`uncompressed`
compression_type	64	Filename character string	Empty string
revision	64	Portable character string	Empty string
size	16	Integer character string	None
source	Undefined	Pathname character string	None

Software_File Common Attributes

The following attributes describe each instance of the software_file class, and are inherited by each instance of the files and control_files classes:

cksum

An integer character string representing a 32-bit cyclic redundancy check (CRC) identical to that returned in the first field of the output of the `cksum` utility, as defined in POSIX.2.

compressed_cksum

Indicates the `cksum` CRC of the compressed software file in the same manner as the *cksum* attribute.

This attribute may be used to verify the integrity of a compressed file, and to help determine if a file to be copied is already present at the target.

compressed_size

Indicates the size of the compressed software file in the same manner as the *size* attribute.

This attribute can be used for computation of disk space analysis when the file will remain compressed after a copy.

compression_state

Indicates which one of the following conditions is true:

- Uncompressed but permitted to be compressed in a distribution (if this attribute has the value `uncompressed` or if no value is supplied for the attribute)
- Already compressed (if this attribute has the value `compressed`)
- Uncompressed and not permitted to be compressed in a distribution (if this attribute has the value `not_compressible`)

compression_type
Specifies the compression method used to compress the file if the value of the ***compression_state*** attribute is `compressed`.

The values supported for ***compression_type*** are implementation defined. The way in which an implementation uses this value to implement or execute the compression or uncompression of a file is undefined.

revision
This is a string indicating the revision level of the file.

size
Indicates the size of the software file in bytes as defined in POSIX.1. *st_size*.

source
When used in a PSF, this attribute specifies the pathname of the file or control_file to be placed in the distribution by the `swpackage` utility.

2.14 Files

Files are the actual files and directories that make up the fileset. Many of the file attributes (such as *owner*, *group*, and *mode*) are derived from, and dependent upon, a POSIX.1 file system.

The file class inherits attributes from the software_file common class.

A particular file object is identified within a fileset by the *path* attribute. When a file is located on a distribution, the *path* attribute indicates the intended installation location of the file. The value of the *path* attribute is also the path below the storage directory for that fileset within file storage structure of the distribution. See Section 5.1.2 on page 128. While a file is installed (in an installed_software object), the *path* attribute indicates the actual location of the file. This path is relative to the root directory for that installed_software object.

For regular files, the value of the *size* attribute is the actual file size in bytes. For symbolic links, this is the string length of the *link_source* attribute. For hard links, directories, and block and character special files, this is always zero. These types are set to zero since the actual space required by these types depends on the file system. An implementation should consider the impact of these types as part of disk space analysis.

The *cksum* attribute only has meaning for a file with type of regular file.

Table 2-13 Attributes of the File Class

Attribute	Length	Permitted Values	Default Value
path	Undefined	Pathname character string	None
archive_path	Undefined	Pathname character string	Empty string
gid	16	Integer character string	Undefined
group	Undefined	Filename character string	Empty string
is_volatile	8	One of: `true`, `false`	`false`
link_source	Undefined	Pathname character string	None
major	16	Portable character string	None
minor	16	Portable character string	None
mode	16	Octal character string	None
mtime	16	Integer character string	None
owner	Undefined	Filename character string	Empty string
type	8	One of: `f`, `d`, `h`, `s`, `p`, `b`, `c`, `x`, `a`,	`f`
uid	16	Integer character string	Undefined

File Attributes

These attributes, along with the attributes listed in Table 2-7 on page 14, describe each instance of the file class:

archive_path

This attribute is used to designate the path to the archive that this file should be added to, instead of installing it to the ‘‘path’’ location. When used in conjunction with the *save_files* option, the .o file that previously existed in the archive is saved, and can be restored.

gid
The numeric group id of the file (POSIX.1. *st_gid*).

group
The group name of the file (POSIX.1. *gr_name*).

is_volatile
A volatile file is a file whose contents can change, or which can be removed after it has been installed.

link_source
The pathname of the target of the link.

This attribute only has meaning if the file type is a hard or symbolic link.

major
This attribute only has meaning if the file type is character or block special file.

The value of this attribute has the same values and meaning as the ***devmajor*** field in the `tar` archive specified in POSIX.1.

minor
This attribute only has meaning if the file type is character or block special file.

The value of this attribute has the same values and meaning as the ***devminor*** field in the `tar` archive specified in POSIX.1.

mode
The mode attribute is an octal representation of the permissions bits of the file (POSIX.1, *st_mode*).

This attribute has no meaning if the file type is a hard or symbolic link.

mtime
This is the time of the last data modification of the file (POSIX.1, *st_mtime*).

owner
The name of the owner of the file (POSIX.1, *pw_name*).

path
The pathname of the file.

type
Supported file types are those described in POSIX.1, file types, plus hard link and symbolic link.

The permitted values of this attribute are the following: `f` (regular file), `d` (directory), `h` (hard link), `s` (symbolic link), `p` (named pipe (FIFO)), `b` (block special device), `c` (character special device), `x` (delete file during an install or update), `a` (archive file during install or update).

uid
The numeric user id of the file (POSIX.1, *st_uid*).

2.15 Control_Files

Control_files can be scripts, data files, or INFO files. The product and fileset INFO files in the software packaging layout are included as control_files. Control scripts are the vendor-supplied scripts executed at various steps by the software administration utilities.

The control_file class inherits attributes from the software_file common class.

A particular control_file object is identified within a product or fileset by the *tag* attribute. The *path* attribute is the storage location of the file relative to the control directory. For distributions, the control directory is the directory in the software packaging layout where the control_files are stored. For installed_software objects, this control directory location is undefined.

Table 2-14 Attributes of the Control File Class

Attribute	Length	Permitted Values	Default Value
tag	64	Filename character string	None

interpreter
path
result
	Undefined
	Undefined
	16
		Filename character string
		Filename character string
		One of: `none`, `success`, `failure`, `warning`
			`sh`
			None
			`none`

Control_File Attributes

These attributes, along with the attributes listed in Table 2-7 on page 14, describe each instance of the control_file class:

interpreter

The name of the interpreter used to execute those control_files that are executed as part of the utilities defined in this Software Administration specification.

Within a distribution, a value for this attribute other than `sh` implies that the distribution is not a conformant one. Such a distribution may be one which is conformant with extensions. See Section 1.3 on page 4.

path

The filename of the control_file.

Multiple control_file entries can have the same value of the *path* attribute. This implies that the same script is executed in different steps within the execution of a utility.

result

Contains the result of the execution of the control script.

This attribute is only valid for control_files in installed_software. A complete list of legal results is contained in Table 2-14.

tag

The identifier of the control_file.

All control files are loaded and maintained within the distribution and installed software catalogs by the utilities defined in this Software Administration specification. These utilities execute control scripts with particular tags at various steps in the execution of the utility. The values for the *control_file* tag attribute for which this Software Administration specification defines behavior are as follows:

```
request
response
checkinstall
preinstall
postinstall
unpreinstall
unpostinstall
verify
fix
checkremove
preremove
postremove
configure
unconfigure
space
```

Chapter 3

Common Definition for Utilities

This chapter defines the common definitions and behaviors of the utilities defined in this Software Administration specification. These utilities conform to the utility syntax guidelines in POSIX.2. The utilities themselves are defined in Chapter 4.

3.1 Synopsis

The following is the general synopsis format for the utilities:

<sw_utility> [-d|-r] [-p] [-u] [-a *attribute*] [-c *catalog*] [-s *source*[
[-f *file*][-t *targetfile*][-x *option=value*][-X *options_files*]
[*software_selections*][@ *targets*]

3.2 Description

The utilities all operate on *software_selections* in source or target software_collections or both.

3.3 Options

Each of the utilities in this standard does not support all of the options shown below. Each utility supports the options indicated in its synopsis section and those indicated after the description of the options in this section. All options can be repeated. Except where otherwise stated within this Software Administration specification, the behavior for repeated options is undefined. In addition to those shown below, the `-W` (capital-W) option is reserved for implementation extensions. See POSIX.2.

`-a` *attribute*

Used to specify the attributes on which the utility operates.

This option can be used multiple times to specify a set of attributes.

Applies to *swlist*, and *swmodify*.

`-c` *catalog*

Used to specify a file with the software definition file syntax or directory with the exported catalog structure.

This is where software catalog information (metadata) is to be stored to or retrieved from. If this information fits into one file, then the catalog can be a file, otherwise it will be a directory. See Chapter 5 and Section 5.2 on page 130.

Applies to *swask*, *swconfig*, *swinstall*, *swlist*, and *swmodify*.

`-d` Indicates to the utility that the operation is on a distribution instead of installed_software.

Applies to *swlist*, *swmodify*, *swremove*, and *swverify*.

`-f` *file*

Reads the list of *software_selections* from *file*.

If this option is specified multiple times, all the software specified by each file is included in the operation. All of the software specified by using this option, as well as all the software

specified directly as arguments to the utility, is included in the operation.

The file contains one software selection per line, where a software selection uses the syntax for `software_spec` defined in Section 3.4.1 on page 38. Blank lines is ignored. Within the file, the `#` (pound) character acts as a comment character. On any line containing a `#` (pound) character, all characters that follow the `#` (pound) character up to, but excluding, the next `<newline>`, is ignored.

Applies to all utilities.

`-p` Previews the operation without making any permanent modifications to the target.

An implementation should run any control scripts that are executed as part of the selection or analysis phase of the command being previewed, but does not run any that are executed in the execution phase.

This option can be used with any or all of the other options to understand the impact of an operation before performing it.

Applies to *swconfig*, *swcopy*, *swinstall*, *swmodify*, *swpackage*, and *swremove*.

`-r` Indicates to the utility that the operation is on an installed_software object located at an alternate root, instead of either a distribution or the installed_software object located at `/`.

Applies to *swinstall*, *swlist*, *swmodify*, *swremove*, and *swverify*.

`-s` *source*
Specifies the software source for the operation.

For *swinstall*, *swask*, and *swcopy* a source can be specified using the syntax in Section 3.4.2 on page 42. For *swpackage*, the source is a product specification file.

Applies to *swask*, *swcopy*, *swinstall*, and *swpackage*.

`-t` *targetfile*
Reads the list of targets from *targetfile*.

If this option is specified multiple times, all the targets specified by each file is included in the operation. All of the targets specified by using this option, as well as all the targets specified directly as arguments to the utility, is included in the operation.

The file contains one target per line, where a target uses the syntax for `software_collection_spec` defined in Section 3.4.2 on page 42. Blank lines is ignored. Within the file, the `#` (pound) character acts as a comment character. On any line containing a `#` (pound) character, all characters that follow the `#` (pound) character up to, but excluding, the next `<newline>`, is ignored.

Applies to *swconfig*, *swcopy*, *swinstall*, *swlist*, *swmodify*, *swpackage*, *swremove*, and *swverify*.

`-u` This is the option used to specify undo or delete behavior to a utility.

Applies to *swconfig* and *swmodify*.

`-x` *option=value*
Used to override the value of an extended option in the defaults file.

The extended options supported are described in Section 3.5.2 on page 44. This option can be specified multiple times. If any extended option is defined more than once, the precedence rules from Section 3.5.3.1 on page 54 is used.

Applies to all utilities.

-X *options_file*

Used to override the defaults specified in the system defaults file.

The options supported are described in Section 3.5.2 on page 44. This option can be specified multiple times. If any extended option from any file is defined more than once, the precedence rules from Section 3.5.3.1 on page 54 is used.

The file has the format defined in Section 3.5.3 on page 52.

Applies to all utilities.

3.3.1 Non-interactive Operation

All utilities except *swask* are by default non-interactive. The *swinstall* and *swconfig* utilities also define interactive modes for executing `request` scripts independent of the *swask* utility.

The way in which *swinstall*, *swcopy* and *swpackage* utilities handle multiple volumes for sources or targets is implementation defined.

3.4 Operands

There are two types of operands that may be specified on the command line, ***software_selections*** and ***targets***. The ***software_selections*** refer to the software objects (bundles, products, subproducts and filesets) to be operated on. The targets refer to the target software_collections where the software selections are applied. These two operand types is separated by the `@` operand. With the exception of ***swpackage***, the behavior of all utilities defined in this Software Administration specification is undefined if no ***software_selections*** are provided.

3.4.1 Software Specification and Logic

The following specifies the syntax for software selections in utilities (`software_spec`) and in dependency specifications (`dependency_spec`). This syntax is applied by the utilities to search a software_collection catalog for software. Note that the tokens shown below are defined in the Glossary. [3]

```
%token    FILENAME_CHARACTER_STRING      /* as defined in Glossary */
%token    NEWLINE_STRING                 /* as defined in Glossary */
%token    PORTABLE_CHARACTER_STRING      /* as defined in Glossary */
%token    SOFTWARE_PATTERN_MATCH_STRING  /* as defined in Glossary */
%token    WHITE_SPACE_STRING             /* as defined in Glossary */


%start  software_selections
%%

software_selections       : software_selections  ws software_spec
                          | software_spec
                          ;

software_spec             : bundle_software_spec
                          | product_software_spec
                          ;

bundle_software_spec      : bundle_qualifier  version
                          | bundle_qualifier  '.' product_qualifier   version
                          ;

bundle_qualifier          : bundle_qualifier  '.' bundle_tag
                          | bundle_tag
                          ;

product_software_spec     : product_qualifier  version
                          ;

product_qualifier         : product_tag  subproduct_qualifier  fileset_qualifier
                          ;

subproduct_qualifier      : /* empty */
                          | subproduct_qualifier  '.' subproduct_tag
                          | '.' subproduct_tag
                          ;
```

3. For examples of the use of specifications in this section, see Appendix A.

```
fileset_qualifier         : /* empty */
                          | '.' fileset_tag
                          ;

bundle_tag                : sw_pattern
                          ;

product_tag               : sw_pattern
                          ;

fileset_tag               : sw_pattern
                          ;

subproduct_tag            : sw_pattern
                          ;

version                   : /* empty */
                          | ',*'
                          | version_qualifier
                          ;

version_qualifier         : version_qualifier  ver_item
                          | ver_item
                          ;

ver_item                  : ',' ver_id '='
                          | ',' ver_id '=' sw_pattern
                          | ',' 'r' rel_op dotted_string
                          ;

sw_pattern                : SOFTWARE_PATTERN_MATCH_STRING
                          ;

ver_id                    : 'r'  |  'a'   | 'v'   | 'l'   | 'q'
                          ;

rel_op                    : '==' | '!=' | '>=' | '<=' | '<' | '>'
                          ;

dotted_string             : FILENAME_CHARACTER_STRING
                          ;

ws                        : WHITE_SPACE_STRING
                          ;



%start  dependency_spec
%%

dependency_spec           : dependency_spec  '|' software_spec
                          | software_spec
                          ;
```

If the `software_spec` identifies a bundle, product or subproduct software object, then all filesets contained within that object are included as part of that specification. For software selections, this means that all of these filesets are included. For dependency specifications, this means that all of these filesets are needed in order to meet the dependency.

If a software_spec identifies a set of filesets that is less that the entire set of filesets within a bundle or product, the software_spec identifies a partial bundle or product.

Only the specified strings is used to generate a software_spec. Blanks do not appear between items. The sw_pattern and dotted_string must be enclosed in quotes if they contain blanks or commas. The bundle_tag, product_tag, subproduct_tag, and fileset_tag consists of one or more characters from the filename character set, with the exception that the following three characters . , : (period, comma, and colon) are not used.

Searching a ***software_collection*** catalog for software using a software_spec yields a list of zero or more software objects that match the software_spec . The rules to be used in the search are the following:

1. The software_spec is compared against software in the software collection. The leftmost sw_pattern of the software_spec is matched against the ***tag*** attribute of all bundles and products in the software collection. All objects that match are initially included for consideration. If the sw_pattern does does not match any bundle or product, no objects are included.

   The version specified in the software_spec is compared against the ***revision***, ***architecture***, ***vendor_tag***, ***location***, and ***qualifier*** attributes of the objects matching the leftmost sw_pattern . If any ver_id in the software_spec does not match its corresponding attribute, that object is removed from consideration. If the same ver_id is given more than once, all the comparisons specified are performed and all must succeed to be considered a match.

   Table 3-1 Software_spec Version Identifiers

ver_id	Attribute
r	revision
a	architecture
v	vendor_tag
l	location
q	qualifier

   An implementation may define additional ver_id items along with the attributes and objects to which they apply.

   For each object still included for consideration, each successive sw_pattern , left to right, is applied to the bundles, products, subproducts and filesets within that object. The same sw_pattern may match multiple bundle, product, subproduct and fileset objects. If any sw_pattern does not match any objects within the current object, the current object is removed from consideration. If a fileset matches a sw_pattern but there is still an unmatched sw_pattern in the software_spec , that fileset is not selected.

   When there are no more sw_patterns left in the software_spec , all the objects identified by the rightmost sw_pattern of the software_spec are included in the list of software that match the software_spec .

2. The comparison performed when the operator is = will be a software pattern match as described in the Glossary. If the ver_id is specified and the value is an empty string, then the comparison is successful only if the corresponding attribute is not specified. See

Section 3.4.1.1 on page 42.

3. When `rel_op` is used,[4] the comparison is performed on the specified attribute by dividing it into segments separated by the `.` (period) character. If there is no period in an attribute, it contains one segment. The segments are compared with the corresponding segments of the `dotted_string`. If all characters in both segments to be compared are decimal digit characters (0-9),[5] the comparison is based on the decimal numeric value of the segments, starting with the leftmost segment. If either segment includes a any character other than a decimal digit character, a string comparison is made to determine the relation. String comparisons is made using, as a collation sequence, the order of characters in If one operand has fewer segments than the other, the unmatched segments is compared against the value `0` (zero).

4. When applied to software in installed software collections, use of either the `l` (location) or `q` (qualifier) `ver_id` causes comparison with the value of the ***location*** or ***qualifier*** attribute respectively for each product or bundle in the installed_software object.

   For distributions, use of either the `l` (location) or `q` (qualifier) `ver_id` is ignored for the purpose of comparisons. Although not used for comparisons, the location and qualifier `ver_ids` are used by the ***swinstall*** utility as the ***location*** attribute for installing the software and the ***qualifier*** attribute for the software respectively.

5. When software selections are applied to a source or target, and a `software_spec` resolves to more than one software object, then the `software_spec` is considered ambiguous. An ambiguous selection may be elective or incidental. An elective ambiguous selection occurs when a `sw_pattern` in a `software_spec` contains a wildcard character or when the version contains a `rel_op`, or when the `sw_pattern` is missing. In all other cases the selection is an incidental ambiguous selection. An incidental ambiguous selection is only valid for ***swlist***, and for other utilities generates an event. (SW_ERROR: SW_SELECTION_NOT_FOUND_AMBIG)

6. If the `software_spec` begins with a ***bundle***tag definition, then that bundle definition is copied or installed with ***swcopy*** or ***swinstall***. Thus, a `software_spec` which matches one or more bundles can be used with all other utilities, but only if they were explicitly installed or copied. However, all subproduct definitions are copied or installed independently of whether they were explicitly selected. Thus, a `software_spec` which matches subproducts can always be used on existing products.

For both bundles and subproducts, if some part of their contents exist, then the selection can be found; therefore, any operation will succeed. If none of their contents exist, then the selection cannot be found; therefore, the operation will fail (for those selections not found) as defined in the individual *Extended Description* section of each utility.

---

4. As defined in the syntax for `ver_item`, this comparison is required for ***revision*** only and any other comparison is undefined.
5. Note that leading zeros are acceptable in such segments.

3.4.1.1 *Fully-qualified Software_spec*

A fully-qualified `software_spec` is one in which no fields contain a shell pattern match string and all version distinguishing attributes are specified as **"ver_id=<*value*>"** (if a value is supplied) or as **"ver_id="** (if no value is supplied or if the value supplied is an empty string). Note that a fully-qualified `software_spec` always identifies a software object unambiguously.

When a `software_spec` is generated by *swlist*, only the following *tag*s are included: the product *tag* for products, the bundle *tag* for bundles, the product and fileset *tag* for filesets, and the product and subproduct *tag* for subproducts.

3.4.1.2 *Software Compatibility*

Products contain attributes (*os_name*, *os_version*, *os_release*, and *machine_type*) related to the *uname*( ) function defined by POSIX.1. These attributes are used by the *swinstall*, *swconfig*, and *swverify* utilities to determine if software is compatible with a target host. A product is considered compatible with a target host if each of the uname attributes of the product contains a pattern in its definition that matches the corresponding values returned by the *uname*( ) function on the target host. If any of these attributes is undefined, it is considered to match any target host. The compatibility test applies to all components of a product, including subproducts and filesets.

Bundles, like products, possess uname attributes. The values of the bundle uname attributes determine the compatibility of the bundle in conjunction with the corresponding attributes of products within the bundle. A product specified as part of a bundle is considered compatible if both the product and bundle uname attributes designate that the software is compatible. As with products, if any of these attributes is undefined, it is considered to match any target host.

3.4.2 Source and Target Specification and Logic

Source and target software_collections are specified using the following syntax.

```
%token          HOST_CHARACTER_STRING      /* as defined in Glossary */
%token          PATHNAME_CHARACTER_STRING  /* as defined in Glossary */
%start  target
%%

target          : software_collection_spec
                ;

software_collection_spec  : HOST_CHARACTER_STRING  ':' PATHNAME_CHARACTER_STRING
                          | HOST_CHARACTER_STRING  ':'
                          | HOST_CHARACTER_STRING
                          | PATHNAME_CHARACTER_STRING
                          ;

%start  source
%%

source          : software_collection_spec
                ;
```

The `:` (colon) is required if both the host and pathname are specified, or if the host portion starts with a `/` (slash). The pathname portion is an absolute path. The colon is not allowed by itself.

The HOST_CHARACTER_STRING portion refers to the implementation defined identifier for a host. If it is not specified, then the local host is assumed.

The PATHNAME_CHARACTER_STRING portion refers to the software_collection *path* attribute (the location on the host of the distribution or installed_software object).

When the PATHNAME_CHARACTER_STRING is not specified for installed_software, the directory / is used.

A PATHNAME_CHARACTER_STRING other than / for an installed_software object is referred to as an alternate root directory. When the PATHNAME_CHARACTER_STRING is not specified for source distributions, the value of the *distribution_source_directory* default option is used. When the PATHNAME_CHARACTER_STRING is not specified for target distributions, the value of the *distribution_target_directory* default option is used.

For installed_software objects, the value of the *installed_software_catalog* option is used to further clarify which installed software object is actually being targeted. Multiple installed_software objects may share the same path attribute, but they have separate catalog information because they are distinct objects. The installed_software *path* attribute prepended to the value of the *installed_software_catalog* option forms the key for the object into the catalog information. Use of the *installed_software_catalog* is independent of the -c option.

An implementation supports source and target distributions in the directory format described in Section 5.3 on page 147, for all utilities. An implementation supports a source distribution in the serial format for *swask*, *swinstall*, and *swcopy* utilities. An implementation supports a target distribution in the serial format for *swlist*, *swcopy*, and *swpackage*. Whether data on an existing target distribution in serial format is overwritten or merged is implementation defined. An implementation need not support a target distribution in the serial format for *swverify*, *swremove*, and *swmodify*. Unless otherwise stated, support for serial distributions includes support for both extended *tar* and extended *cpio* archives. See Section 5.3 on page 147. The format of these archives is defined in POSIX.1.

3.5 External Influences

3.5.1 Defaults and Options Files

The defaults file allows setting of system wide defaults for extended options that define information (location of files and other objects), behavior, and policy control items for the utilities defined in this Software Administration specification. The location of the defaults file is implementation defined. An implementation may define separate defaults files for each task. These options also may be specified for each user in the manager role in the file `$HOME/.swdefaults`.

3.5.2 Extended Options

The utilities in this Software Administration specification support the following extended options as noted. If a default value is defined, it is listed after the `=` (equal sign).

allow_downdate=false

Controls the ability to replace a fileset with one of a lower revision.

If *allow_downdate* false, do not allow installation of a lower revision of a fileset that is already installed at a higher revision in this location.

If *allow_downdate* true, allow installation of a lower revision of a fileset.

Applies to *swinstall*.

allow_incompatible=false

Controls the ability to install software that is not compatible with the underlying operating system, as defined in Section 3.4.1.2 on page 42.

If *allow_incompatible* false, do not allow incompatible software to be operated on if the installed_software path is `/`.

If *allow_incompatible* true, then attempt the operation.

Applies to *swinstall*, *swconfig*, and *swverify*.

allow_multiple_versions=false

Controls the ability to configure multiple versions of a product.

If *allow_multiple_versions* false, do not attempt to configure a second version of a fileset if one is already configured. If *allow_multiple_versions* true, then attempt the operation.

Applies to *swconfig*.

ask=false

Controls the ability to execute `request` scripts for selected software.

If *ask=false*, the utilities do not run any `request` scripts for selected software. The behavior of *swask* for *ask=false*, is undefined.

If *ask=true*, the utilities execute all `request` scripts for selected software after resolving selections, but before initiating analysis on the targets. This is the default value for *swask*.

If *ask=as_needed*, the utilities execute any `request` scripts for selected software that does not already have a `response` file in the control directory where the script would be executed. The location of this control directory depends on whether the `-c` option has been set.

Applies to *swask*, *swconfig*, and *swinstall*.

autoreboot=false

Controls automatic rebooting of the target host. If *autoreboot=false*, do not automatically reboot the target host, even if a fileset installed requires a reboot to take effect.

If *autoreboot=true*, automatically reboot the target host if a fileset requiring a reboot is installed.

Applies to *swinstall*.

autorecover=false

Controls automatic recovery if an error occurs during install, as specified in section *swinstall* on page 100.

If *autorecover=false* and an install error occurs, no error recovery is provided at all, not even as an extension to this Software Administration specification. Consequently, no attempt should be made to restore the original state of the system prior to install. The value of the fileset *state* attribute is set to `corrupt`.

If *autorecover=true* and an error occurs, then for any fileset having an install error, implementations will execute the unpostinstall script (if the postinstall script had been run) and the unpreinstall script,[6] restore the files within the fileset from a copy saved prior to the failed install, and the restore the value of the *state* attribute. After recovery of the applicable filesets, installation can continue with the rest of the filesets in that product and the rest of the products in the software selections.

Applies to *swinstall*.

autorecover_product

Like the *autorecover* option, this option will recover a complete product to the previous state, including running the appropriate *unpreinstall* and *unpostinstall* scripts, if the product postinstall fails.

This option differs from the *autorecover* option in that the product will be recovered to its previous state if any fileset has a script error or a file loading error. This means the install results in either a fully updated product or the previous product. With the POSIX *autorecover* option, only that fileset is recovered, and the other filesets are attempted, resulting in a partially updated product.

autoselect_dependencies=as_needed

Controls automatic dependency selection.

If *autoselect_dependencies=true*, (the default for all utilities except *swinstall* and *swcopy*), prerequisite and corequisite dependencies are autoselected if possible during the selection phase. Autoselection of a dependency is done using the software selection logic found in Section 3.4.1 on page 38. These dependencies are then operated on as if they were selected explicitly.

If *autoselect_dependencies=as_needed*, (the default for *swinstall* and *swcopy*), then autoselected dependencies is only operated upon if the dependency is not already met on the target. This value only applies to *swcopy* and *swinstall*.

If *autoselect_dependencies=false*, then no dependencies are autoselected for operation. For install and copy, if the dependencies are not already met on the target, an error occurs when

6. Since failure prior to executing the preinstall script should have no side effects, a failure implies that the unpreinstall script requires execution.

enforce_dependencies=true.

Applies to *swask*, *swconfig*, *swcopy*, *swinstall*, and *swverify*.

autoselect_dependents=false
Controls automatic dependency selection.

If *autoselect_dependents=true* , dependent software (software that depends on this software) will be autoselected if possible during the selection phase. This dependent software is operated upon unless the dependency can be met by other software on the target. If dependent software exists that can not still meet its dependencies through other unselected software, then an error occurs.

If *autoselect_dependents=false*, no dependent software is autoselected.

Applies to *swconfig*, and *swremove*.

check_contents=true
Controls verification of file contents.

If *check_contents=true*, then *swverify* checks the *mtime*, *size* and *cksum* attributes of files.

If *check_contents=false*, then *swverify* does not check the attributes.

This applies to both distribution and installed_software files.

Applies to *swverify*.

autoselect_patches=true
This option causes all patches (except for those that are *superseded* by other patches) that have an *ancestor* attribute defined for selected software to be automatically selected as well during *swinstall* and *swcopy* operations. The default setting for this option is `true`.

Applies to *swinstall* and *swcopy*.

check_permissions=true
Controls verification of file permissions.

If *check_permissions=true*, then *swverify* checks the *owner*, *uid*, *group*, *gid*, *mode* attributes of files, and the *major* and *minor* attributes of device files.

If *check_permissions=false*, then *swverify* does not check the attributes.

This only applies to installed_software files.

Applies to *swverify*.

check_requisites=true
Controls verification of fileset requisites.

If *check_requisites=true* then *swverify* checks the *prerequisites*, *corequisites*, and *exrequisites* attributes of files.

If *check_requisites=false*, then *swverify* does not check the attributes.

This applies to both distribution and installed_software.

Applies to *swverify*.

check_scripts=true
Controls the running of the `verify` script.

If *check_scripts=true*, then *swverify* runs the vendor supplied `verify` script for each fileset when operating on installed_software objects. When the *F* option of *swverify* is used, the

vendor supplied `fix` script is also executed.

If *check_scripts=false*, then *swverify* does not run the scripts.

Applies to *swverify*.

check_volatile=false

Controls check of volatile files.

If *check_volatile=true*, then *swverify* includes files whose *is_volatile* attribute is set to `true` in its check of files and their attributes.

If *check_volatile=false*, then *swverify* does not include volatile files. This is useful to eliminate potentially ''spurious'' reports from *swverify* when the only file changes are those to files known in advance to be volatile.

Applies to *swverify*.

compress_files=false

Controls whether uncompressed files are to be compressed in the target distribution, as specified by the value of *compression_type*.

If *compress_files=true*, then all files except those that have a *compression_state* of `not_compressible` are compressed, or remain compressed.

If *compress_files=false*, uncompressed files are not compressed, and the status of any compressed file is determined by the value of *uncompress_files*.

Applies to *swcopy*.

compression_type=implementation_defined_value

Specifies the compression type used to compress the software files.

The values supported for *compression_type* are implementation defined.

The way in which an implementation uses this value to implement or execute the compression or uncompression of a file is undefined.

Applies to *swcopy*.

defer_configure=false

Controls automatic configuration at install.

If *defer_configure=false*, software being installed is also configured when the root directory is `/`.

If *defer_configure=true*, then the software is installed but not configured, and may require configuration (using *swconfig*) before being used.

Applies to *swinstall*.

defer_deleting_files=false

This option defers the deleting files of type "delete file" by during install or update.

Applies to *swinstall*.

distribution_source_directory=implementation_defined_value

Specifies the default distribution directory.

When a source specification does not contain a path specification, the value of this extended option is used as as the default source distribution directory. When a source specification does contain a path specification, it is used.

Applies to *swask*, *swcopy*, and *swinstall*.

distribution_target_directory=implementation_defined_value
Specifies the default distribution target.

When a target specification does not contain a path specification, the value of this extended option is used as the default distribution target. When a target specification does contain a path specification, it is used. For *swpackage*, this is used only when *media_type=directory*.

Applies to *swcopy*, *swlist*, *swmodify*, *swpackage*, *swremove*, and *swverify*.

distribution_target_serial=implementation_defined_value
Specifies the default distribution target.

When a target specification does not contain a path specification and *media_type=serial*, the value of this extended option is used as the default distribution target. When a target specification does contain a path specification, it is used.

Applies to *swpackage*.

enforce_dependencies=true
Controls the enforcement of dependency specifications.

If *enforce_dependenciesr=true*, no utility except *swremove* and the unconfigure option of *swconfig* proceeds unless necessary dependencies have been selected, or already exist in the proper state on the target. The *swremove* utility and the unconfigure portion of the *swconfig* utility does not proceed if operating on the selected software leaves dependent software with their dependencies unresolved beyond what existed before the utility was executed.

If *enforce_dependencies=false*, then all utilities proceed even if some dependencies are not met. Enforcement of dependencies is independent of whether or not they were autoselected.

Applies to *swconfig*, *swcopy*, *swinstall*, *swremove*, and *swverify*.

enforce_dsa=true
Controls the handling of disk space analysis errors.

If *enforce_dsa=true*, the implementation defined error handling procedure is invoked when the disk space analysis indicates there is not enough disk space.

If *enforce_dsa=false*, then the operation is attempted even if disk space analysis indicated a problem.

Applies to *swcopy*, *swinstall*, and *swpackage*.

enforce_locatable=true
Controls the handling of errors when relocating a non-relocatable fileset.

If *enforce_locatable=true*, an error is generated if an attempt is made to relocate a non-relocatable fileset.

If *enforce_locatable=false*, an attempt is made to relocate the fileset in any case.

Applies to *swinstall*, and *swverify*.

enforce_scripts=true
Controls the handling of errors generated by scripts.

If *enforce_scripts=true*, the implementation defined error handling procedure is invoked when the vendor supplied scripts return an error.

If *enforce_scripts=false*, all script errors is treated as warnings, and the utility attempts to continue operation.

Applies to *swinstall*, and *swremove*.

files=

Lists the pathnames of file objects to be added or deleted.

If *files='file1 file2 file3 ...'*, then catalog information for those files is added or deleted. When files are added, the attributes of the file are retrieved from the actual file on the installed file system. File objects being added or deleted can also be specified in the `INFO` file format. There is no supplied default.

Applies to *swmodify*.

follow_symlinks=false

Controls the following of symbolic links

If *follow_symlinks=false*, then do not follow any symbolic links that may exist in the packaging source.

If *follow_symlinksr=etrue*, then attempt to follow symbolic links.

Applies to *swpackage*.

installed_software_catalog=implementation_de fined_value

Specifies installed software catalog.

This extended option, along with the installed_software *path* attribute, defines the logical installed_software object upon which the utility is operating. This extended option is resolved relative to the `PATHNAME_CHARACTER_STRING` portion of the *targets* operand. See Section 3.4.2 on page 42.

This option allows an implementation to define where the catalog information is stored. This option also allows multiple logical installed_software objects to share the `PATHNAME_CHARACTER_STRING` where the software is installed.

Applies to *swask*, *swconfig*, *swinstall*, *swlist*, *swmodify*, *swremove*, and *swverify*.

logfile=implementation_defined_value

Specifies the location of the the logfile for the management role.

Logfile structure for all roles, logfile locations for other roles, and the effect of this option on logfile location is implementation defined.

Applies to all utilities except *swlist*.

loglevel=1

Controls the amount of output sent by the utility to log files (not to stdout and stderr).

See Section 3.6.5 on page 69.

Applies to all utilities except *swlist*.

match_target=false

When set to `true`, this option adds software items from the source depot to the selection list that have the same ancestor as software already installed, if the ancestor software is installed in only a single location.

Applies to *swinstall*.

media_capacity=0

The storage capacity in megabytes of the output media.

A value of 0 (zero) indicates an infinite capacity.

Applies to *swpackage*.

media_type=directory
The default media type.

If *media_type=directory*, the distribution is located in the value of the *distribution_target_directory* option.

If *media_type=serial*, the distribution is located in the value of the *distribution_target_serial* option.

Applies to *swpackage*.

one_liner=implementation_defined_value
Specifies attributes to list.

The *one_liner* option specifies the attributes to list by default when neither `v` and `a` attribute options are specified. Only attributes that apply to each object listed are included for that object. At least one of the *tag* attribute (of products, subproducts, filesets and control scripts) or the *path* attribute (of files) is included. The order of attributes in the output listing need not be the order of the attributes specified in this option. The listing format used by *one_liner* is undefined.

Applies to *swlist*.

patch_commit=false
When set to `true`, this removes the saved files for the patches specified in the software selections operands, meaning that patch can no longer be rolled back.

Applies to *swmodify*.

patch_save_files=true
This option allows the user to save existing files, enabling rollback of patches, or to not save files (meaning that patches can not be rolled back).

Applies to *swinstall*.

*patch_filter=**
This option can only be used in conjunction with the options *autoselect_patches* or *patch_match_target*. This option is used to filter the automatically selected patches to only those meeting the tag, version and category filtering criteria specified. The default value of this option is ''*'' (all patches).

Applies to *swinstall* and *swcopy*.

patch_match_target=false
This option corresponds to the update *match_target* option, automatically selecting patches that have ancestors defined for software currently installed or in the depot. Patches that are superseded by other patches are not included in the selections. The default value of this option is `false`.

Applies to *swinstall* and *swcopy*.

reconfigure=false
Controls reconfiguring of software.

If *reconfigure=false*, do not reconfigure software if it is already in the `configured` state.

If *reconfigure=true*, reconfigure the software even if it is already in the `configured` state.

Applies to *swconfig*.

recopy=false

Controls copying of filesets.

If *recopy=false*, do not copy a fileset that is already available on the target at the same version.

If *recopy=true*, then copy the fileset in any case.

Applies to *swcopy*.

reinstall=false

Controls reinstallation of filesets.

If *reinstall=false*, do not install a fileset that already has the same version already installed.

If *reinstall=true*, then reinstall the fileset even if this version is already installed.

Applies to *swinstall*.

reinstall_files=false

When set to `true`, this option directs the utility to reinstall (or recopy) each file independent of whether the file is already installed (or in the depot) correctly. When set to `false`, if the file is already installed (based on mtime, size and cksum), it is not reinstalled (or retransferred over the network for distributed install).

Applies to *swinstall*, *swcopy* and *swpackage*.

reinstall_files_use_cksum=true

This option is only applicable if *reinstall_files=false*. When set to `true`, this option directs the utility to include a checksum check when determining if the file is up to date. When set to `false`, only size and mtime are checked.

Applies to *swinstall*, *swcopy* and *swpackage*.

save_modified_files=false

This option compares the catalog information for each file in each fileset that is being updated against the actual size and cksum attributes of the installed files, and saves those files that differ. Where these files are saved is implementation defined.

Applies to *swinstall*.

saved_files_directory=<implementation_defined_value>

This option allows the user to specify where the saved files are retained. This directory is also stored in the installed software catalog for a *patch* fileset, so the correct location can be identified when removing or committing a patch, or updating a patches' ancestor.

This value should be an absolute path. If the value is relative or an empty string, the behavior is undefined.

Applies to *swinstall*.

select_local=true

Controls default selection of target.

If *select_local=true*, and no targets are specified, then the local host is selected as the target.

If *select_local=false*, then the local host is not automatically included.

Applies to all utilities except *swask* and *swpackage*.

software=

Specifies a default set of `software_selections` for the utility.

Applies to all utilities in this Software Administration specification.

targets=

Specifies a default set of *targets* for the utility.

See the *select_local* option.

Applies to all utilities in this Software Administration specification except *swpackage*.

uncompress_files=false

Controls whether compressed files are to be uncompressed in the target distribution, as specified by the value of the *compression_type* attribute of the file.

If *uncompress_files=false*, all files with a *compression_state* attribute value of `compressed` remain compressed, and the status of uncompressed files is determined by the value of *compress_files*.

If *uncompress_files=true*, all compressed files are uncompressed before being written to the target distribution.

Applies to *swcopy*.

verbose=1

Controls the amount of output sent by the utility to stdout and stderr, but not to log files.

For values which are non-negative integers, an increase in *verbose* does not decrease the information sent stdout and stderr. All implementations support the values `0` (zero) and `1` (one). If *verbose=0*, nothing is written to either stdout or stderr. The effect of other values of *verbose* is undefined. See also Section 3.6.3 on page 69, and Section 3.6.4 on page 69.

Applies to all utilities in this Software Administration specification.

3.5.3 Extended Options Syntax

The syntax is the same for options specified on the command line and for those specified in the options file. Individual options use this syntax:

```
%token    FILENAME_CHARACTER_STRING  /* as defined in Glossary */
%token    PORTABLE_CHARACTER_STRING  /* as defined in Glossary */
%token    SHELL_TOKEN_STRING         /* as defined in Glossary */
%token    WHITE_SPACE_STRING         /* as defined in Glossary */


%start  software_option
%%

software_option          : command_qualifier  keyword '=' value
                         ;

command_qualifier        : /* empty */
                         | command '.'
                         ;

value                    : multi_value
                         | single_value
                         ;

multi_value              : value ws single_value
                         | single_value
                         ;
```

```
single_value             : SHELL_TOKEN_STRING
                         ;

command                  : FILENAME_CHARACTER_STRING
                         ;

keyword                  : FILENAME_CHARACTER_STRING
                         ;

ws                       : WHITE_SPACE_STRING
                         ;
```

With respect to this syntax, the following apply:

- *command*
  A **keyword** prefixed by the *command* name applies to that utility only.

  If no prefix exists, then the **keyword** applies to all the utilities that support it.

- **keyword**
  Names the option or operand being defined, for example, *allow_incompatible* for *swinstall*, and *verbose* for all the utilities.

  The allowable characters for a **keyword** are as defined in the filename character set, plus the – (hyphen) character.

- **value**
  Assigns the value to the **keyword**.

  All extended options are single valued except those that contain lists of `software_specs` or `software_collection_specs` . Quoting of strings and escaping of characters is handled as specified in POSIX.2.

When specified on the command line, multiple option specifications can be included after a single `-x` option if included in quotes and separated by white space. Multiple `-x` options can also be used.

For option and defaults files, blank lines and all comment text are ignored. Comment text is any sequence of characters beginning with a `#` (pound) character which is neither escaped nor quoted, and continuing through the end of that line.

If the white space between single values contains a `<newline>` , either it is ‘‘escaped’’ or the entire value is quoted.

The following are examples of this syntax:

```
loglevel=1
allow_incompatible=false
autoselect_dependencies="as_needed"
software="Foo,r=1.2,a=hp-ux  bar,a=Aix_3.2"
targets="hosta:/  hostb hostc:"

software="Foo,r=1.2,a=hp-ux
bar,a=Aix_3.2"

targets="hosta:/
         hostb
         hostc:
         "
```

3.5.3.1 *Precedence for Option Specification*

Multiple option or operand specifications have a precedence that defines which specifications are used.

Only the option specifications with the highest level of precedence are used for each options and operands. The precedence is in increasing order:

1. System defaults file
2. User defaults file
3. Options file
4. Command line options and operands

If there are multiple instances of options at any particular level, then the following rules apply:

- If both **keyword** and *command*.**keyword** exist in the set of defaults or options files for this level, the *command* uses the latter, more specific, definition.
- All values for *software* and *targets* options from all levels are all included in the resulting *software_selections* and *target_selections* for the command.
- For options besides *software* and *targets*, the behavior when multiple or conflicting specifications are made is undefined. This rule applies to options such as `s` *source* where implementations may choose to assign a logical interpretation to multiple source specification. The same rule applies to options that are mutually exclusive.

3.5.4 Input Files

The definitions for the *swpackage* utility (see *swpackage* on page 111) and the *swmodify* utility (see *swmodify* on page 108) respectively specify additional input files specific to those utilities.

3.5.5 Access and Concurrency Control

An implementation of this Software Administration specification allows a user to create, modify, delete, and access a catalog that describes a software object located where it is permissible for that user to respectively create, modify, delete, and access files. Other authorization, authentication and concurrency control requirements and mechanisms are undefined within this Software Administration specification. This Software Administration specification does provide event definitions that an implementation can use for access and concurrency control errors.

If the user of a utility does not have the proper authorization to run a utility, access a software_collection, or access software objects within that utility, the target may generate an event (SW_ERROR: SW_ACCESS_DENIED).

If the concurrency control mechanism prevents simultaneous operation on a software collection or software object, the target may generate an event (SW_ERROR: SW_CONFLICTING_SESSION_IN_PROGRESS).

If the command will proceed anyway, then the target may generate an event (SW_WARNING: SW_CONFLICTING_SESSION_IN_PROGRESS).

If the concurrency control mechanism fails for other reasons, the target may generate an event (SW_ERROR: SW_SOC_LOCK_FAILURE).

3.5.6 Environment Variables

Environment variables are a feature of this Software Administration specification inherited from POSIX.1. The following environment variables affects the execution of all the utilities defined in this Software Administration specification:

LANG
This variable determines the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale.

See POSIX.2.

LC_ALL
This variable determines the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE
This variable determines the interpretation of sequences of bytes of text data as characters (for example, single versus multibyte characters in values for vendor defined attributes).

LC_MESSAGES
This variable determines the language in which messages should be written.

LC_TIME
This variable determines the format of dates (*create_date* and *mod_date*) when displayed by *swlist*.

It should also be used by all utilities when displaying dates and times in stdout, stderr, and logging.

TZ
This variable determines the time zone for use when displaying dates and times.

3.6 External Effects

3.6.1 Control Script Execution and Environment

The utilities defined in this Software Administration specification causes control files to be interpreted according to the following rules:

1. If no value is set for the *control_file.interpreter* attribute, or if the value is set to either the empty string or `sh`, the script is interpreted by the POSIX.2 shell.
2. If the value of *control_file.interpreter* is set to a value other than the empty string or `sh`, then the utility determines the availability of the interpreter in an operating system dependent fashion equivalent to searching **PATH** for an executable file with a filename equivalent to the value of *control_file.interpreter*. If the interpreter is determined to be available, the control file is interpreted using that interpreter.
3. If no interpreter is available, then a return code value of `1` (one) is presumed for the script, and all other actions defined for that return code are assumed. See the definitions for *swask* on page 78, *swconfig* on page 81, *swinstall* on page 90, *swremove* on page 115, and *swverify* on page 121.

During the execution of each such control script, the following environment variables are defined for the environment of the control_script:

SW_CATALOG
The value of the *installed_software.catalog* attribute indicating the location or identification of the catalog relative to the **SW_ROOT_DIRECTORY**.

SW_CONTROL_DIRECTORY
The directory where the executing script is located.

This directory is readable from within control script execution and is writable from commands within control scripts when the `request` script is being executed. All `control_files` are readable by any control script.

SW_CONTROL_TAG
The *tag* of the script being executed.

This allows the control_script to tell what *tag* is being executed when the actual script path is defined for more than one *tag*.

SW_LOCATION
The base directory where the product or fileset will be installed or is already installed.

This is the value of the *location* attribute.

SW_PATH
A **PATH** which, at least, contains all utilities defined by the POSIX.2.[7]

SW_ROOT_DIRECTORY
The *installed_software.path* attribute of the installed software object within which the software containing this control_file is installed.

This is the directory relative to which all operations with the script are performed.[8]

---

7. POSIX.2 requires and defines the C function *confstr*() that obtains such a **PATH**.
8. For example, if this normally has a value of `/`, but if a proxy install is done to a target directory `/mnt/test/`, this will have the value of `/mnt/test/`.

SW_SESSION_OPTIONS

The pathname of a file containing the value of every option defined for the software utility being executed, using the options syntax described in Section 3.5.3 on page 52.

The option syntax is restricted such that the command prefix is not used, there are no spaces on either side of the = (equal sign), and multiple valued options have the values quoted.

This environment variable allows scripts to retrieve any options and values for this command other than the ones provided explicitly via environment variables. When the file pointed to by **SW_SESSION_OPTIONS** is made available to `request` scripts, the ***targets*** option contains a list of `software_collection_specs` for all targets specified for the command. When the file pointed to by **SW_SESSION_OPTIONS** is made available to other scripts, the *targets* option contains the single `software_collection_spec` for the targets on which the script is being executed.

An implementation should ensure that each `software_collection_spec` contained in the value of the *targets* option is the same between invocations of commands. This will help ensure that any per-target information stored by the `request` script can be located by the subsequent scripts.

SW_SOFTWARE_SPEC

The value of the fully-qualified `software_spec` identifying the software object containing this control script.

In order to allow a control script to know whether it is a new install or an update in order to change its behavior accordingly, the ***swinstall*** utility sets the environment variable SW_ANCESTORS to the fully qualified ***software_specs*** of the ancestor or ancestors of the fileset that currently are installed.

See Section 3.4.1.1 on page 42.

3.6.1.1 Control Script Stdout and Stderr

The scripts may send information, particularly about reasons for error conditions to stdout and stderr. The utilities logs stdout and stderr to the logfile of the role executing the script.

3.6.1.2 Control Script Return Code

The scripts return with a return code of 0 (zero), 1 or 2. Additionally, `checkinstall` , `checkremove` , `configure` , and `unconfigure` scripts may return with a return code of 3. The return codes 4 through 31 (inclusive) are reserved for future use. The meaning of these return codes is shown in the following table:

Table 3-2 Script Return Codes

Return Code	Effect of Return Code	Status
0	The script executed successfully. The utility will proceed normally.	SW_NOTE
1	The script had an error. The utility generates an error event and implement the error procedure defined for this script type.	SW_ERROR
2	The script had a warning. The utility will generate a warning event and continue.	SW_WARNING
3	The script is forcing a deselection of this product or fileset. The utility will generate a note and skip this product or fileset during any further processing.	SW_NOTE
4-31	Reserved.	

All scripts, with the exception of the `request` script, are non-interactive.

An implementation can define behaviors for additional script return codes. Any such behavior is implementation defined.

Return codes with no behavior defined by either this Software Administration specification or the implementation should be treated using the behavior associated with return code 2.

3.6.2 Asynchronous Events

The following are the set of events generated by the utilities defined in this Software Administration specification. These events are generated during the course of a execution of a utility. See Section 3.6.5 on page 69.

The event codes and their numeric values are listed in Table 3-4, Table 3-5, Table 3-6, and Table 3-7, inclusive.[9]

Each event generated also has a severity status associated with it. The event status that can occur for each event is also listed. Event status also has a numeric value, as described in Table 3-3, Table 3-4, Table 3-5, Table 3-6, and Table 3-7. In addition, all numeric values between 0 and 255 (inclusive) are reserved, either for use in this Software Administration specification (as described in the accompanying tables) or for use in future revisions of this Software Administration specification.

9. Not all events are generated by each utility. For example, events related to script execution only apply to *swinstall*, *swask*, *swremove* and *swverify*. The specific events generated by each utility are defined in the section for that utility.

Table 3-3 Event Status

Status	Effect of Event	Value
SW_NOTE	The operation continues normally	0
SW_ERROR	Implementation defined error handling procedure is invoked	1
SW_WARNING	The operation continues normally	2

A command will not have an exit code of zero if any `SW_ERROR` event occurred during the course of a command.

The descriptions in the following tables describe the conditions that lead to this event, and the set of possible event status values for the event. The tables also include ''Manager info'' and ''Target info'' that describe the additional information that may be logged for manager and target role event logging respectively. See Section 3.6.5 on page 69.

Table 3-4 lists general source and target role events. The way in which some of these events are generated (if at all) may be different for different implementations.

Table 3-4 General Error Events

Event Code	Event Status	Description	Value
SW_ILLEGAL_STATE_TRANSITION	SW_ERROR	The manager is requesting a phase out of order. Manager info: target. Target info: current phase.	1
SW_BAD_SESSION_CONTEXT	SW_ERROR	The manager has contacted the wrong target, or this is not a valid manager for this session. Manager info: target. Target info: information about the initiator.	2
SW_ILLEGAL_OPTION	SW_ERROR	An illegal or unrecognized option was sent. Manager info: target, number of options. Target info: option names and values.	3
SW_ACCESS_DENIED	SW_ERROR	The user has insufficient privilege to perform the requested operation. Manager info: target. Target info: information about the initiator.	4
SW_MEMORY_ERROR	SW_ERROR	The target role had a memory allocation error (for example, out of swap). Manager info: target. Target info: reasons for error.	5
SW_RESOURCE_ERROR	SW_ERROR	The target role had a resource allocation error such as maximum number of processes reached, maximum number of files open, etc. Manager info: target. Target info: reasons for error.	6
SW_INTERNAL_ERROR	SW_ERROR	The target role had an internal implementation error. Manager info: target. Target info: reasons for error.	7
SW_IO_ERROR	SW_ERROR	An I/O error occurred while performing this command. Manager info: target. Target info: reasons for error.	8

Table 3-5 lists the source and target role events related to initialization of a session and ending a session. The way in which some of these events are generated (if at all) may be different for different implementations.

Table 3-5 Session Events

Event Code	**Event Status**	**Description**	**Value**
SW_AGENT_INITIALIZATION_FAILED	SW_ERROR	Failed to initialize a target session. Manager info: target. Target info: reasons for error.	10
SW_SERVICE_NOT_AVAILABLE	SW_ERROR	The target role is not accepting new requests. Manager info: target. Target info: reasons for error.	11
SW_OTHER_SESSIONS_IN_PROGRESS	SW_WARNING	There are other sessions in progress that may affect the results of this command. Manager info: target, number of sessions. Target info: information about other sessions.	12
SW_SESSION_BEGINS	SW_NOTE	The command begins on the target. Manager info: target. Target info: information about the initiator of the command.	28
SW_SESSION_ENDS	SW_NOTE SW_WARNING SW_ERROR	The command ends on the target successfully, with warnings, or with errors. Manager info: target. Target info: none.	29
SW_CONNECTION_LIMIT_EXCEEDED	SW_ERROR	The limit of source or target role sessions on this host has already been reached. Manager info: target, number of sessions. Target info: number of sessions, limit.	30
SW_SOC_DOES_NOT_EXIST	SW_ERROR	The requested target or source software collections does not exist. Manager info: target. Target info: reasons for error.	31
SW_SOC_IS_CORRUPT	SW_ERROR	The software_collection exists, but the information is corrupt. Manager info: target. Target info: reasons for error.	32
SW_SOC_CREATED	SW_NOTE	The target software_collection did not previously exist and was created. Manager info: target. Target info: none.	34
SW_CONFLICTING_SESSION _IN_PROGRESS	SW_ERROR SW_WARNING	A conflicting session is in progress that will prevent this operation (error), or cause its results to possibly be invalid (warning). Manager info: target. Target info: information about other sessions.	35

Event Code	Event Status	Description	Value
SW_SOC_LOCK_FAILURE	SW_ERROR	Can not set the proper access control to this source or target. Manager info: target. Target info: reasons for error.	36
SW_SOC_IS_READ_ONLY	SW_ERROR SW_NOTE	The software_collection is a read only source for a read source or target (note), or is a target to be modified (error). Manager info: target. Target info: none.	37
SW_SOC_IS_REMOTE	SW_ERROR SW_NOTE	The software_collection is on a remote file system. (Whether note or error is implementation defined). Manager info: target. Target info: none.	38
SW_SOC_INCORRECT_MEDIA_TYPE	SW_ERROR	The distribution is an incorrect type for the command (for example, a tape for *swremove*). Manager info: target. Target info: reasons for error.	39
SW_SOC_IS_SERIAL	SW_NOTE	The distribution has a serial format (for example, a tape). Manager info: target. Target info: none.	40
SW_SOC_INCORRECT_TYPE	SW_ERROR	The software_collection is of the wrong type (distribution or installed_software) for the operation. Manager info: target. Target info: target type.	41
SW_CANNOT_OPEN_LOGFILE	SW_ERROR	Cannot open logfile to log the software_collection events. Manager info: target. Target info: reasons for error.	42
SW_SOC_AMBIGUOUS_TYPE	SW_ERROR	The software collection is inadequately specified for the operation. Manager info: target. Target info: reason for error	49
SW_TERMINATION_DELAYED	SW_NOTE	The target role is currently analyzing or executing a command and will terminate the session once completed. Manager info: target. Target info: none.	50
SW_CANNOT_INITIATE_REBOOT	SW_WARNING	The target role failed to initiate the reboot operation of an install command and requires manual reboot. Manager info: target. Target info: reasons for error.	51

Table 3-6 lists the source and target role events related to the analysis phase of the commands. Some of these are also related to the execution phase of the commands.

Table 3-6 Analysis Phase Events

Event Code	Event Status	Description	Value
SW_ANALYSIS_BEGINS	SW_NOTE	The analysis phase begins on the target. Manager info: target. Target info: none.	52
SW_ANALYSIS_ENDS	SW_NOTE SW_WARNING SW_ERROR	The analysis phase ends on the target. The analysis may have succeeded, had warnings, and/or errors. Manager info: target. Target info: none.	53
SW_EXREQUISITE_EXCLUDE	SW_NOTE	One or more filesets were excluded automatically as the software identified as exrequisites was also specified to be selected. Manager info: target, number of filesets. Target info: `software_specs`	56
SW_CHECK_SCRIPT_EXCLUDE	SW_NOTE	One or more checkinstall or checkremove scripts have caused the software to be unselected and excluded from further processsing. Manager info: target, number of filesets. Target info: software_specs	57
SW_CONFIGURE_EXCLUDE	SW_NOTE	One or more configure or unconfigure scripts have caused the software to be unselected and excluded from further processsing. Manager info: target, number of filesets. Target info: `software_specs`	58
SW_SELECTION_IS_CORRUPT	SW_ERROR	The software selection was found, but its state was `corrupt` or `transient`. Manager info: target, number of selections Target info: `software_specs`	59
SW_SOURCE_ACCESS_ERROR	SW_ERROR	Failure contacting or retrieving information from the source. Manager info: target. Target info: reasons for error.	60
SW_SOURCE_NOT_FIRST_MEDIA	SW_ERROR	The source does not have a media number of `1` (needed for retrieval of the `INDEX`). Manager info: target, current media number. Target info: current media number.	61
SW_SELECTION_NOT_FOUND	SW_ERROR SW_NOTE	One or more software selections can not be found. This is an error for install or copy; otherwise, a note. Manager info: target, number of selections. Target info:	62

Event Code	Event Status	Description	Value
		software_specs	
SW_SELECTION_NOT _FOUND_RELATED	SW_ERROR SW_NOTE	One or more software selections can not be found as specified, but another version exists. This is an error for install or copy; otherwise, a note. Manager info: target, number of selections. Target info: software_specs	63
SW_SELECTION_NOT _FOUND_AMBIG	SW_ERROR	One or more software selections can not be unambiguously determined. Manager info: target, number of selections. Target info: software_specs	64
SW_FILESYSTEMS_NOT_MOUNTED	SW_ERROR SW_WARNING	One or more file systems on the target are not mounted. Manager info: target, number of file systems. Target info: file system names.	65
SW_FILESYSTEMS_MORE_MOUNTED	SW_WARNING	One or more file systems mounted are not in file system table. Manager info: target, number of file systems. Target info: file system names.	66
SW_HIGHER_REVISION_INSTALLED	SW_ERROR SW_WARNING	One or more filesets have a higher revision already installed. Whether error or warning is controlled by *allow_downdate* option. Manager info: target, number of filesets. Target info: software_specs	67
SW_NEW_MULTIPLE_VERSION	SW_ERROR SW_NOTE	One or more products would create a new version in an installation. Whether error or warning is controlled by *allow_multiple_versions* option. Manager info: target, number of products. Target info: software_specs	68
SW_EXISTING_MULTIPLE_VERSION	SW_ERROR SW_WARNING SW_NOTE	The command is operating on an existing multiple version of one or more products. If trying to install two versions into one location, generate an event. Warning or note controlled by *allow_multiple_versions* option. Manager info: target, number of products. Target info: software_specs	69
SW_DEPENDENCY_NOT_MET	SW_ERROR SW_WARNING	One or more dependencies can not be met. Whether error or warning is controlled by *enforce_dependencies* option. Manager info: target, number of filesets. Target info: software_specs, dependency_specs	70

Event Code	**Event Status**	**Description**	**Value**
SW_NOT_COMPATIBLE	SW_ERROR SW_WARNING	One or more products are incompatible for this target. Whether error or warning is controlled by *allow_incompatible* option. Manager info: target, number of products. Target info: `software_specs`	71
SW_CHECK_SCRIPT_WARNING	SW_WARNING	One or more `checkinstall`, `checkremove` or `verify` scripts had a warning. Manager info: target, number of filesets. Target info: `software_specs`	72
SW_CHECK_SCRIPT_ERROR	SW_ERROR	One or more `checkinstall`, `checkremove` or `verify` scripts failed. Manager info: target, number of filesets. Target info: `software_specs`	73
SW_DSA_INTO_MINFREE	SW_ERROR SW_WARNING	Disk space analysis is over the minimum free limit, but not the overall limit on the target. Whether error or warning is controlled by *enforce_dsa* option. Manager info: target, number of file systems. Target info: file system names, amount over the minimum free.	74
SW_DSA_OVER_LIMIT	SW_ERROR SW_WARNING	Disk space analysis is over the absolute limit. Whether error or warning is controlled by *enforce_dsa* option. Manager info: target, number of file systems. Target info: file system names, amount over the limit.	75
SW_DSA_FAILED_TO_RUN	SW_ERROR SW_WARNING	Disk space analysis had an internal error and failed to run. Whether error or warning is controlled by *enforce_dsa* option. Manager info: target, number of filesets. Target info: `software_specs`	76
SW_SAME_REVISION_INSTALLED	SW_NOTE	One or more filesets have the same revision and are being reinstalled because *reinstall*true or *recopy*true. Manager info: target, number of filesets. Target info: `software_specs`	77
SW_ALREADY_CONFIGURED	SW_NOTE	One or more filesets are already configured Whether they are reconfigured is controlled by *reconfigure* option. Manager info: target, number of filesets. Target info: `software_specs`	78
SW_SKIPPED_PRODUCT_ERROR	SW_NOTE	One or more filesets will be skipped because of another error within their product. (Error handling is implementation defined). Manager	79

Event Code	Event Status	Description	Value
		info: target, number of filesets. Target info: `software_specs`	
SW_SKIPPED_GLOBAL_ERROR	SW_NOTE	One or more filesets will be skipped because of a global error (such as disk space analysis failure) within the analyze phase. (Error handling is implementation defined). Manager info: target, number of filesets. Target info: `software_specs`	80
SW_FILE_IS_REMOTE	SW_WARNING SW_NOTE	One or more files would be created or removed on a remote file system. (Policy for loading remote files is implementation defined). Manager info: target, number of files. Target info: file paths.	81
SW_FILE_IS_READ_ONLY	SW_WARNING	One or more files will not be attempted to be created or removed on a read only file system. Manager info: target, number of files. Target info: file paths.	82
SW_FILE_NOT_REMOVABLE	SW_WARNING	One or more files could not be removed (for example, text busy, or non-empty directories). Manager info: target, number of files. Target info: file paths.	83
SW_FILE_WARNING	SW_WARNING	One or more files had warnings in analysis or execution. Manager info: target, number of files. Target info: file paths.	84
SW_FILE_ERROR	SW_ERROR	One or more files had errors in analysis or execution. Manager info: target, number of files. Target info: file paths.	85
SW_NOT_LOCATABLE	SW_WARNING SW_ERROR	A fileset is not locatable. Controlled by the *enforce_locatable* option. Manager info: target, number of filesets. Target info: `software_specs`	86
SW_SAME_REVISION_SKIPPED	SW_NOTE	One or more filesets have the same revision and are being skipped because *reinstall*false or *recopy*false. Manager info: target, number of filesets. Target info: `software_specs`	87

Table 3-7 lists the error, warning and notes for target role events related to the execution phase.

Table 3-7 Execution Phase Events

Event Code	Event Status	Description	Value
SW_EXECUTION_BEGINS	SW_NOTE	The execution phase begins on the target. Manager info: target. Target info: none.	88
SW_EXECUTION_ENDS	SW_NOTE SW_WARNING SW_ERROR	The execution phase ends on the target. Manager info: target. Target info: none.	89
SW_SELECTION_SKIPPED _IN_ANALYSIS	SW_NOTE	One or more selections will not be included for execution because they were determined to be skipped in analysis. Manager info: target, number of filesets. Target info: `software_specs`	90
SW_SELECTION_NOT_ANALYZED	SW_ERROR	One or more software selections were found, but were not analyzed. Manager info: target, number of filesets. Target info: `software_specs`	91
SW_WRONG_MEDIA_SET	SW_ERROR	The source media current being used is not the same as that used for analysis. Manager info: target. Target info: information about current media and needed media.	92
SW_NEED_MEDIA_CHANGE	SW_NOTE	The target needs the next media. (Interactive support for media change is implementation defined). Manager info: target, needed media sequence number. Target info: needed media sequence number.	93
SW_CURRENT_MEDIA	SW_NOTE	The current media sequence number for the target Manager info: target, current media sequence number. Target info: current media sequence number.	94
SW_PRE_SCRIPT_WARNING	SW_WARNING	One or more `preinstall` , `preremove` , `unpreinstall` , or `fix` scripts had a warning. Manager info: target, number of scripts. Target info: `software_spec` , script *tag*	95
SW_PRE_SCRIPT_ERROR	SW_WARNING SW_ERROR	One or more `preinstall` , `preremove` , `unpreinstall` , or `fix` scripts failed. Manager info: target, number of scripts. Target info: `software_spec` , script *tag*	96
SW_FILESET_WARNING	SW_WARNING	One or more filesets had a warning. Manager info: target, number of filesets. Target info:	97

Event Code	Event Status	Description	Value
		`software_specs`	
SW_FILESET_ERROR	SW_ERROR	One or more filesets had an error. Manager info: target, number of filesets. Target info: `software_spec s`	98
SW_POST_SCRIPT_WARNING	SW_WARNING	One or more `postinstall` , `postremove` , or `unpostinstall` scripts had a warning. Manager info: target, number of filesets. Target info: `software_specs`	99
SW_POST_SCRIPT_ERROR	SW_WARNING SW_ERROR	One or more `postinstall` , `postremove` , or `unpostinstall` scripts failed. Manager info: target, number of filesets. Target info: `software_specs`	100
SW_POSTKERNEL_WARNING	SW_WARNING	The `postkernel` kernel build script had a warning. Manager info: target. Target info: reasons for error.	101
SW_POSTKERNEL_ERROR	SW_ERROR	The kernel failed to build. Manager info: target. Target info: reasons for error.	102
SW_CONFIGURE_WARNING	SW_WARNING	One or more `configure` or `unconfigure` scripts had a warning. Manager info: target, number of scripts. Target info: `software_specs`, script *tag*	103
SW_CONFIGURE_ERROR	SW_WARNING SW_ERROR	One or more `configure` or `unconfigure` scripts failed. Manager info: target, number of scripts. Target info: `software_specs`, script *tag*s	104
SW_DATABASE_UPDATE_ERROR	SW_ERROR	An update to the catalog information for installed_software or distributions failed. Manager info: target. Target info: reasons for error.	105
SW_REQUEST_WARNING	SW_WARNING	One or more `request` scripts had a warning. Manager info: `software_specs`	106
SW_REQUEST_ERROR	SW_ERROR	One or more `request` scripts failed. Manager info: `software_specs`	107
SW_COMPRESSION_FAILURE	SW_ERROR	A file could not be compressed or uncompressed. Manager info: target. Target info: filepath.	112
SW_FILE_NOT_FOUND	SW_ERROR	File is missing from the source or target software_collection. Manager info: target, number of files. Target info: file path.	113

Event Code	Event Status	Description	Value
SW_FILESET_BEGINS	SW_NOTE	The execution phase of a fileset begins. Manager info: target. Target info: `software_spec`	117
SW_CONTROL_SCRIPT_BEGINS	SW_NOTE	The execution of a control script begins. Manager info: target. Target info: `software_spec` , control script *tag*	118
SW_FILE_BEGINS	SW_NOTE	The execution phase of file begins. Manager info: target. Target info: file path.	119

3.6.3 Stdout

Events with a status of `SW_NOTE` will, if permitted by the value *verbose*, be directed to stdout. Manager role events will, if permitted by the value *verbose*, be directed to stdout. Nothing is written to stdout if *verbose*0 (zero). The writing of any target role events to stdout is undefined.

3.6.4 Stderr

If any events with a status of `SW_ERROR` or `SW_WARNING` occur on a target role, this information is communicated to the management role. In addition, at least a single message for that target will, if permitted by the value *verbose*, be directed to the stderr of the management role. Nothing is written to stderr if *verbose*0 (zero).

The sending of any additional messages to stderr of the management role is undefined.

See *verbose* in section Section 3.5.2 on page 44.

3.6.5 Logging

The management role and target role each log events. The way in which logging is implemented, including the location of the logfiles, is implementation defined.

Which messages, if any, are placed in the source role logfile is undefined.

All implementations support the values `0` (zero), `1` (one) and `2..` If *loglevel*0, nothing is written to log files. The target role logs all events, except file level events, when *loglevel*1. The target role redirects, to the logfile, stderr and stdout from control scripts when *loglevel*1. When *loglevel*2, the target role logs file level events. For values which are non-negative integers, an increase in *loglevel* does not decrease the information logged for a given role. All other behavior regarding logging is undefined.

See *loglevel* in section Section 3.5.2 on page 44.

3.7 Extended Description

See the individual utility sections for the complete extended descriptions of each task. This section lists the steps common to the utilities.

There are three phases in the utilities. Targets may be processed in any order or in parallel.

1. Selection phase
   When the user specifications are resolved, including source, target, and software selections.

2. Analysis phase
   When the utility attempts to discover conditions which may cause failure when operating on the selected software.

3. Execution phase
   When the actual operations on the software objects take place.

When a utility initiates a session on a target, generate an event.
(SW_NOTE: SW_SESSION_BEGINS)

When the session completes, generate an event.
(SW_NOTE: SW_SESSION_ENDS)

3.7.1 Selection Phase

This section summarizes the common tasks in the selection phase. Errors and warnings are listed along with the tasks.

3.7.1.1 Starting a Session

On invocation, each command processes options as defined in Section 3.5.3.1 on page 54.

The command exits if the user does not have appropriate privilege. Since implementation of the security scheme is unspecified within this Software Administration specification, appropriate privilege is implementation defined. An implementation may generate the following events at any point in the execution of the command if they are applicable to that implementation:

(SW_ERROR: SW_ACCESS_DENIED)
(SW_ERROR: SW_ILLEGAL_OPTION)
(SW_ERROR: SW_MEMORY_ERROR)
(SW_ERROR: SW_RESOURCE_ERROR)
(SW_ERROR: SW_INTERNAL_ERROR)
(SW_ERROR: SW_TERMINATION_DELAYED)
(SW_ERROR: SW_IO_ERROR)

3.7.1.2 Specifying Targets

A target is specified using the syntax in Section 3.4.2 on page 42. Each target passes the following validation checks. If any of these checks fail, the command invokes the implementation defined error handling procedure.

- If the target role was unable to initialize the session on the target host, generate an event.
  (SW_ERROR: SW_AGENT_INITIALIZATION_FAILED)
- If administrative access is denied by the target role on the target, generate an event.
  (SW_ERROR: SW_ACCESS_DENIED)
- Except for *swinstall*, *swcopy*, and *swpackage*, if the target does not exist on a host, an error is generated:

(SW_ERROR: SW_SOC_DOES_NOT_EXIST)

- For *swinstall*, *swcopy*, and *swpackage*, if the target directory does not exist on a host, it is created with default attributes.
(SW_NOTE: SW_SOC_CREATED)

- If the target is corrupt, generate an event.
(SW_ERROR: SW_SOC_IS_CORRUPT)

- If the target is the wrong type (installed_software or distribution) for the target type the user specified (with a `-r` or `-d` option respectively), generate an event.
(SW_ERROR: SW_SOC_INCORRECT_TYPE)

  If both a distribution object and an installed_software object exist at the location specified in the target, and neither the `-d` nor the `-r` option is specified, generate an event.
  (SW_ERROR: SW_SOC_AMBIGUOUS_TYPE)

- If the target is a serial distribution, generate an event.
(SW_NOTE: SW_SOC_IS_SERIAL)

  If the command is *swcopy* or *swpackage*, a serial distribution is overwritten by default. If the command is *swremove*, *swmodify*, or *swverify*, and the implementation does not support these on a serial distribution, an event is generated.
  (SW_ERROR: SW_SOC_INCORRECT_MEDIA_TYPE)

- If the target is not able to open the implementation defined logfile, generate an event.
(SW_ERROR: SW_CANNOT_OPEN_LOGFILE)

- If the operation needs to modify the target, and it is on a read-only media or file system, generate an event.
(SW_ERROR: SW_SOC_IS_READ_ONLY)

- An implementation may generate the following events, if they are applicable to that implementation, when validating a target:
(SW_ERROR: SW_SERVICE_NOT_AVAILABLE)
(SW_WARNING: SW_OTHER_SESSIONS_IN_PROGRESS)
(SW_ERROR: SW_CONNECTION_LIMIT_EXCEEDED)
(SW_NOTE: SW_SOC_IS_REMOTE)
(SW_ERROR: SW_FILESYSTEMS_NOT_MOUNTED)
(SW_ERROR: SW_FILESYSTEMS_MORE_MOUNTED)

3.7.1.3 *Specifying the Source*

This section only applies to *swcopy* and *swinstall*. The source contains software organized in the software packaging layout. A target can be specified using the syntax in Section 3.4.2 on page 42.

If source is specified, then it is resolved in the context of the management role. If source is not specified, then a default value is supplied as defined in Section 3.5.3.1 on page 54.

A source must satisfy the following validation checks:

- Verify that the source exists, and may be accessed
(SW_ERROR: SW_SOURCE_ACCESS_ERROR).

- If administrative access is denied by the source role for that source, generate an event.
(SW_ERROR: SW_ACCESS_DENIED)

- Obtain information on what software is available from the source. If the information cannot be retrieved, or if an problem occurs while processing it, generate an event.

(SW_ERROR: SW_SOURCE_ACCESS_ERROR)

- If the source is a serial media, and the *media*sequence_number is not 1, then generate an event. (Only the first media has the catalog information on it).
(SW_ERROR: SW_SOURCE_NOT_FIRST_MEDIA)

3.7.1.4 *Software Selections*

Software selections can be specified on the command line or in an input file using the syntax in Section 3.4.1 on page 38.

3.7.2 Analysis Phase

This section summarizes the common operations and events in the analysis phase. The analysis phase occurs before the execution phase, and involves executing checks to determine whether or not the execution should be attempted.

When the analysis phase begins, generate an event
(SW_NOTE: SW_ANALYSIS_BEGINS)

To begin the analysis phase, the management role (the host on which the utility was invoked) communicates the selection information to each target in the target list. The target role accesses the source (for *swinstall* or *swcopy*) or target (for other utilities) to get the product catalog information for the software selections. The product catalog information includes control script information in the *control_files* attribute of filesets within each product.

- An implementation may generate any of the following events, if they are applicable to that implementation, when attempting the analysis or execution phase.
(SW_ERROR: SW_AGENT_INITIALIZATION_FAILED)
(SW_ERROR: SW_ILLEGAL_STATE_TRANSITION)
(SW_ERROR: SW_BAD_SESSION_CONTEXT)
- If the target role cannot access the source, generate an event.
(SW_ERROR: SW_SOURCE_ACCESS_ERROR)
- If an implementation supports access control for particular operations for particular software objects, and if access is denied for any software object, generate an event.
(SW_ERROR: SW_ACCESS_DENIED)
- If a fileset has an error for which there is not a more specific event defined, generate the generic event.
(SW_ERROR: SW_FILESET_ERROR)
- If a fileset has a warning for which there is not a more specific event defined, generate the generic event.
(SW_WARNING: SW_FILESET_WARNING)
- An implementation may generate the following events as applicable to the error handling procedures for that implementation:
(SW_NOTE: SW_SKIPPED_PRODUCT_ERROR)
(SW_NOTE: SW_SKIPPED_GLOBAL_ERROR)
(SW_WARNING: SW_FILE_IS_READ_ONLY)

See each utility section for the analysis operations specific to each utility. When the analysis phase ends, generate an event.
(SW_NOTE: SW_ANALYSIS_ENDS)

3.7.3 Execution Phase

This section summarizes the common operations and events in the execution phase.

When the execution phase begins, generate an event.
(SW_NOTE: SW_EXECUTION_BEGINS)

The execution phase proceeds through the steps defined for each utility. The following events are common to all utilities.

- When the execution phase executes a script, generate an event.
(SW_NOTE: SW_CONTROL_SCRIPT_BEGINS)
- When the execution phase begins the key operation on a fileset (such as loading or removing), generate an event.
(SW_NOTE: SW_FILESET_BEGINS)
- When the execution phase begins the key operation on a file (such as loading or removing), generate an event.
(SW_NOTE: SW_FILE_BEGINS)
- If at any time there is an error rebuilding any catalog files, generate an event.
(SW_ERROR: SW_DATABASE_UPDATE_ERROR)
- If a fileset has an error for which there is not a more specific event defined, generate the generic event.
(SW_ERROR: SW_FILESET_ERROR)
- If a fileset has a warning for which there is not a more specific event defined, generate the generic event.
(SW_WARNING: SW_FILESET_WARNING)
- For *swinstall* and *swcopy* from a distribution that spans multiple media, an implementation may generate the following events to convey needed media change information. An implementation may, but need not, provide such support for other utilities.
(SW_ERROR: SW_WRONG_MEDIA_SET)
(SW_NOTE: SW_NEED_MEDIA_CHANGE)
(SW_NOTE: SW_CURRENT_MEDIA)
- An implementation may generate the following events for software that will not be executed due to analysis results for that software.
(SW_NOTE: SW_SELECTION_SKIPPED_IN_ANALYSIS)
(SW_ERROR: SW_SELECTION_NOT_ANALYZED)
- The default behavior for filesets in the *removed* state is the same as for filesets that are *non-existent*. Implementations that support the *removed* state should define extensions to the 1387.2 utilities providing operations on *removed* filesets.

See each utility section for the execution operations specific to each utility. When the execution phase completes, generate an event.
(SW_NOTE: SW_EXECUTION_ENDS)

3.7.3.1 Fileset State Transitions

A conforming implementation must maintain the *state* attribute of each fileset to identify the condition and validity of that package. A conforming implementation must use these and only these states as valid values of the *fileset*state attribute.

configured
Indicates the fileset in an installed_software object has been configured.

This state applies to filesets in installed_software objects.

installed
Indicates that the specified fileset has been installed successfully.

corrupt
Indicates that the most recent attempt to handle the fileset was not successful and any recovery actions that were attempted were similarly unsuccessful.

Software can transition from this state via the *swinstall*, *swcopy*, or *swremove* utilities. Other implementation defined methods may also exist for transitioning from this state.

This state applies to filesets in distributions and installed_software objects.

removed
This state indicates that the files for the fileset has been removed but the information remains.

The default behavior when removing software with *swremove* is to also remove its information (metadata) from the catalog. An implementation may define a means for removing software, but maintaining the catalog information. The catalog information can be separately removed with the *swmodify* utility.

This state applies to filesets in distributions and installed_software objects.

available
Indicates the fileset is present in the distribution and may be operated on (copied, installed, etc.) using the appropriate utilities.

This state applies to filesets in a distribution.

transient
Indicates that the fileset is currently being acted on by one of the utilities that modify software files, thus the state of the software is not well defined.

This state should be replaced by another before the utility completes. Presence of this state in the software_collection when no utility is running indicates that a utility was previously interrupted (power failure, kill, etc) and was not able to record a final state indication into the software_collection catalog. In such case, the implementation defined recovery procedures can be used to restore the product to a another state.

This state applies to filesets in distributions and installed_software objects.

3.8 Exit Status

The utility returns one of the following exit codes:

Table 3-8 Return Codes

Return Code	Definition
0	The utility was successful on all targets
1	The utility failed on all targets
2	The utility failed on some targets

The exit status for the *swpackage* utility is different since it is not a distributed utility.

3.9 Consequences of Errors

Utilities can operate on multiple software objects contained in multiple targets. Whether an error impacts a particular software object, all software objects in the target, or all targets, is implementation defined. One exception to this is the minimum error recovery procedure described in *swinstall* on page 100, which describes fileset level recovery during install.

3.10 Error Conditions

The conditions leading to errors are described in Section 3.6.2 on page 58, and Section 3.7 on page 70.

Chapter 4

Software Administration Utilities

The Software Administration Utilities defined in this chapter must be implemented in all conformant systems.

The common definitions for utilities are specified in Chapter 3. Where applicable, the definition for each utility in this chapter refers to these common definitions, to avoid repetition.

- *swask*
- *swcopy*
- *swconfig*
- *swinstall*
- *swlist*
- *swmodify*
- *swpackage*
- *swremove*
- *swverify*

NAME

swask — ask for user responses

SYNOPSIS

`swask [-c` *catalog*`][-f` *file*`][-s` *source*`][-t` *targetfile*`][-x` *option=value*`]`
`[-X` *options_file*`][`*software_selections*`][@` *targets*`]`

DESCRIPTION

The *swask* utility runs the interactive software `request` scripts for the software objects selected. These scripts store the responses to the `response` files for later use by the *swinstall* and *swconfig* utilities.

The *swinstall* and *swconfig* can also run the interactive `request` scripts directly.

OPTIONS

The *swask* utility supports the following options. Where there is no description, the description in Chapter 3 applies.

`-c` *catalog*

Specifies the pathname to an exported catalog structure below which the `response` files created by the `request` scripts are stored for later use.

If the `c` *catalog* option is omitted, the utility uses the source catalog, *<distribution path>*>`/catalog`, as the directory in which to store the `response` files. Hence these `response` files will be the ones available for use by the control scripts executed by *swinstall*.

`-f` *file*

`-s` *source*

`-t` *targetfile*

`-x` *option=value*

`-X` *options_file*

OPERANDS

The *swask* utility supports the *software_selections* operand described in Chapter 3.

Specifying values for the *targets* operand does not imply operations on the target role. The values are simply written as a list of targets for the *targets* option in the file made available to the *request* script via the **SW_SESSION_OPTIONS** environment variable. See Section 3.6.1 on page 56.

EXTERNAL INFLUENCES

See Chapter 3 for descriptions of external influences common to all utilities.

Extended Options

The *swask* utility supports the following extended options. The description in Chapter 3 applies.

autoselect_dependencies=true

distribution_source_directory=implementation_defined_value

distribution_source_serial=implementation_defined_value

logfile=implementation_defined_value

loglevel=1

ask=true

software=

verbose=1

EXTERNAL EFFECTS

See Chapter 3.

EXTENDED DESCRIPTION

See Chapter 3 for general information. There are two phases in the *swask* utility:

1. Selection phase
2. Execution phase

Selection Phase

The software selections apply to the software available from the distribution source if the `-s` option was specified. Otherwise, the software selections apply to software that has been already installed on the targets. Each specified selection is added to the selection list after it passes the following checks:

- If the `-s` option is specified and the selection is not available from the source, generate an event. If the `-s` option is not specified and the selection is not available from the target, generate an event.
  [SW_ERROR: SW_SELECTION_NOT_FOUND]
- If a unique version can not be identified, generate an event.
  [SW_ERROR: SW_SELECTION_NOT_FOUND_AMBIG]

Add any dependencies to the selection list if *autoselect_dependencies=true*.

Execution Phase

For each selected software that has a `request` control script:

- If it does not already exist, build the necessary control directories of the exported catalog structure to hold the `response` file for that software object.[10]
- Set the value of the **SW_CONTROL_DIRECTORY** environment variable to the directory below which the request script writes the `response` file. It may be set to the control directory where the `response` file will eventually be held, or it may be set to another, temporary, directory.

  If a response file can be found from one of the following sources, searched in the order specified, then the implementation ensures that the response file is contained within the directory pointed to by **SW_CONTROL_DIRECTORY**. The means for doing this (e.g., copy, link, symlink) is undefined.

  1. If `-c` *catalog* was specified, any `response` file already existing below that catalog
  2. If `-s` was not specified, any `response` file from the catalog of the target or targets specified

---

10. If there are multiple product versions selected, they has separate control directories as specified in the software packaging layout. Update the global `INDEX` in the exported catalog structure so the appropriate version can be identified later.

3. If `-s` was specified, any `response` file already existing in the source catalog.

- If ***ask=true***, execute the `request` script. If ***ask=as_needed***, execute the `request` script only if a `response` file does not already exist in the control directory.
- The `request` script writes a single `response` file in the control directory[11] defined by the supplied environment variable **SW_CONTROL_DIRECTORY**.
- If a `request` script returns an error and ***enforce_scripts=true***, generate an event and invoke the implementation defined error handling procedures.
  [SW_ERROR: SW_REQUEST_ERROR]

  If a `request` script returns an error and ***enforce_scripts=false***, generate an event.
  [SW_WARNING: SW_REQUEST_ERROR]

  If a `request` script returns a warning, generate an event.
  [SW_WARNING: SW_ASK_SCRIPT_WARNING]

EXIT STATUS

See Chapter 3.

CONSEQUENCES OF ERRORS

See Chapter 3.

11. The *swinstall* and *swconfig* utilities will distribute the `response` file to the targets for use by that targets control scripts.

NAME

swconfig — configure software

SYNOPSIS

```
swconfig [-p][-u][-c catalog][-f file][-t targetfile][-x option=value]
         [-X options_file][software_selections][@ targets]
```

DESCRIPTION

The *swconfig* command configures, unconfigures and reconfigures installed software on the target *hosts* specified on the command line for execution on those hosts.

Configuration primarily involves executing vendor-supplied `configure` and `unconfigure` scripts. These scripts configure or unconfigure the installed software. They are only executed on the target hosts that are intended to actually run the software. Software can be configured more than once by rerunning the `configure` scripts.

Configuration can also be done as part of the *swinstall* and *swremove* utilities.

OPTIONS

The *swconfig* utility supports the following options. Where there is no description, the description in Chapter 3. applies.

`-c` *catalog*

If this option is specified, then use the exported catalog structure at this path as the source of the `response` files.

If *ask=true* or *ask=as_needed*, the control directories in the exported catalog structure are used for both the eventual source of the `response` files, and the control directory where the `request` scripts are executed in order to create any needed `response` files.

`-f` *file*

`-p`

`-t` *targetfile*

`-u` Undo configuration.

This option tells the *swconfig* utility to unconfigure the software, instead of configuring it.

`-x` *option=value*

`-X` *options_file*

OPERANDS

The *swconfig* utility supports the *software_selections* and *targets* operands described in Chapter 3.

EXTERNAL INFLUENCES

See Chapter 3 for descriptions of external influences common to all utilities.

Extended Options

The *swconfig* utility supports the following extended options. The description in Chapter 3 applies.

allow_incompatible=false

allow_multiple_versions=false

ask=false

autoselect_dependencies=true

autoselect_dependents=false

enforce_dependencies=true

installed_software_catalog=implementation_de fined_valuea

logfile=implementation_defined_value

loglevel=1

reconfigure=false

select_local=true

software="

targets=

verbose=1

EXTERNAL EFFECTS

See Chapter 3.

EXTENDED DESCRIPTION

See Chapter 3 for general information. There are three key phases in the *swconfig* utility:

1. Selection phase
2. Analysis phase
3. Execution phase

Selection Phase

Software selections apply to the software installed on the target. Each specified selection is added to the selection list after it passes the following checks:

- If the selection is not found, generate an event.
  [SW_WARNING: SW_SELECTION_NOT_FOUND]
- If the selection is not found at that product location but it does exist in another location, generate an event.
  [SW_WARNING: SW_SELECTION_NOT_FOUND_RELATED]

Add any dependencies to the selection list if *autoselect_dependencies=true* and the task is configure. Add any dependents to the selection list if the *autoselect_dependents=true* and the task is unconfigure.

If *ask=true* then execute the `request` scripts for the selected software as described in the *swask* utility. See *swask* extended description.

Analysis Phase

The following checks are made:

- If configuring, check for compatibility of selections, and if the software is not compatible, generate an event. If *allow_incompatible=true* generate an event. See Section 3.4.1.2 on page 42.
  [SW_WARNING: SW_NOT_COMPATIBLE]

  If *allow_incompatible=false* generate an event.
  [SW_ERROR: SW_NOT_COMPATIBLE]

- If configuring and the state is `corrupt` or `transient`, generate an event.
  [SW_ERROR: SW_SELECTION_IS_CORRUPT]
- If the state is already configured when configuring, or not configured when unconfiguring, generate an event. This event is independent of whether the software will be reconfigured or not, which in turn is controlled by the *reconfigure* option.
  [SW_NOTE: SW_ALREADY_CONFIGURED]
- Check whether configuring this software will create a new configured version of a fileset, and generate an event if it will. See the definition for *version* in the Glossary. If *allow_multiple_versions=true* generate an event.
  [SW_WARNING: SW_NEW_MULTIPLE_VERSION]

  If *allow_multiple_versions=false* generate an event.
  [SW_ERROR: SW_NEW_MULTIPLE_VERSION]
- If the dependencies are not in the `configured` state and have not been autoselected to be configured, generate an event. If *enforce_dependencies=false* generate an event.
  [SW_WARNING: SW_DEPENDENCY_NOT_MET]

  If *enforce_dependencies=true*, generate an event.
  [SW_ERROR: SW_DEPENDENCY_NOT_MET]

Execution Phase

The sequential relationship between the **configure** operations is shown in the following list. Products are ordered by prerequisite dependencies if any. Fileset operations are also ordered by any prerequisites.

- Configure each product
  1. Run `configure` script for the product.
  2. Configure each fileset in the product.
     a. Run the `configure` script for the fileset.
     b. Update the result of the script (control_file). Update the state of the fileset to `configured` in the database for the installed_software object.

If there is no `configure` script for a fileset, the state of the fileset is still changed as specified above.

For **unconfigure** operations, the order of the products and filesets within the products is the reverse of the order of products and filesets for configure. The operations are:

- Unconfigure each product:
  1. Unconfigure each fileset in the product:
     a. Run the `unconfigure` script for the fileset.
     b. Update the result of the script. Update the state of the fileset in the product to `installed` in the database for the installed_software object.
  2. Run the `unconfigure` script for the product.

If there is no `unconfigure` script for a fileset the state of the fileset is still changed as specified above.

Executing Configure Scripts

In this operation, *swconfig* executes vendor-supplied `configure` or `unconfigure` scripts.

The `configure` scripts are interactive; however, they may access the `response` file generated by an interactive `request` script. If any `response` file has been generated, it will exist in the directory pointed to by **SW_CONTROL_DIRECTORY**.

- If a `configure` script returns an error and *enforce_scripts=true*, generate an event and invoke the implementation defined error handling procedures.
  [SW_ERROR: SW_CONFIGURE_ERROR]
- If a `configure` script returns an error and *enforce_scripts=false*, generate an event.
  [SW_WARNING: SW_CONFIGURE_ERROR]
- If the `configure` script returns a warning, generate an event.
  [SW_WARNING: SW_CONFIGURE_WARNING]
- If a `configure` script has a return code of 3, generate an event, and exclude the fileset (or all filesets within the product for a product level script) from any state change between `configured` and `installed`).
  [SW_NOTE: SW_CONFIGURE_EXCLUDE]

EXIT STATUS

See Chapter 3.

CONSEQUENCES OF ERRORS

See Chapter 3.

NAME

swcopy — copy distribution

SYNOPSIS

```
swcopy [-p][-f file][-s source][-t targetfile][-x option=value]
       [-X options_file][software_selections][@ targets]
```

DESCRIPTION

The *swcopy* utility copies or merges software from any source distribution to a target distribution on the local host, or to the *targets* specified on the command line. The distribution can then be accessed by the *swinstall* utility as a source.

OPTIONS

The *swcopy* utility supports the following options. Where there is no description, the description in Chapter 3 applies.

`-f` *file*

`-p`

`-s` *source*

`-t` *targetfile*

`-x` *option=value*

`-X` *options_file*

OPERANDS

The *swcopy* utility supports the *software_selections* and *targets* operands described in Chapter 3.

Whether data on an existing target distribution in serial format is overwritten or merged is implementation defined.

EXTERNAL INFLUENCES

See Chapter 3 for descriptions of external influences common to all utilities.

Extended Options

The *swcopy* utility supports the following extended options. The description in Chapter 3 applies.

autoselect_dependencies=as_needed

autoselect_patches=true

compress_files=false

compression_type=implementation_defined_value

distribution_source_directory=implementation_defined_value

distribution_target_directory=implementation_defined_value

enforce_dependencies=true

enforce_dsa=true

logfile=implementation_defined_value

loglevel=1

*patch_filter=**

patch_match_target=false

recopy=false

reinstall_files=false

reinstall_files_use_cksum=true

select_local=true

software=

targets=

uncompress_files=false

verbose=1

EXTERNAL EFFECTS

See Chapter 3.

EXTENDED DESCRIPTION

See Chapter 3 for general information. There are three key phases in the copy utility:

1. Selection phase
2. Analysis phase
3. Execution phase

Selection Phase

If a software selection has dependency specifications on other software and *autoselect_dependencies=true*, this software is automatically selected using the method described in Section 3.4.1 on page 38. The resulting selection must be unambiguous. This automatically selected software is then copied along with the rest of the selected software. If *autoselect_dependencies=if_needed*, then automatically selected software is only copied if the dependency is not already met.

The *swcopy* utility supports corequisite and prerequisite dependencies for autoselection, but treats them equally (no ordering considerations like *swinstall*). The *swcopy* utility ignores exrequisites.

For *swcopy* each selection added to the selected software list satisfies the following validation checks. If any of these checks result in an error, the selection is not added to the list.

- If the selection is not available from the source, generate an event.
  [SW_ERROR: SW_SELECTION_NOT_FOUND]
- If a unique version can not be identified, generate an event.
  [SW_ERROR: SW_SELECTION_NOT_FOUND_AMBIG]

No compatibility filtering is done for *swcopy*.

In general, usage of the *patch_filter* and *patch_match_target* options for copying patches corresponds exactly to their use in *swinstall*. One difference is that the ‘‘highest superseding’’ rule is not used; instead, all levels of patches are included.

Analysis Phase

The target role uses the file size information obtained from the source to determine whether or not the copy utility proceeds on the given target. If any target fails any of the analysis operations, what software is attempted to be copied is determined by the implementation

defined error handling procedures.

The target role checks the following requirements:

- If the target distribution does not exist on a host, create the path to the target with default attributes and generate an event.
  [SW_NOTE: SW_SOC_CREATED]
- Check that selected filesets are not the same version as already available. If *recopy=false*, note that these filesets will be skipped by generating an event.
  [SW_NOTE: SW_SAME_REVISION_SKIPPED]

  If *recopy=true*, note that they will be recopied by generating an event.
  [SW_NOTE: SW_SAME_REVISION_INSTALLED]
- Verify that the needed dependencies of the filesets are met. If *enforce_dependencies=true*, generate an event.
  [SW_ERROR: SW_DEPENDENCY_NOT_MET]

  If *2enforce_dependencies=false"*, generate an event.
  [SW_WARNING: SW_DEPENDENCY_NOT_MET]
- Check that there is enough free disk space on the target file system to copy the selected products. If there is not enough disk space and *enforce_dsa=true*, generate an event.
  [SW_ERROR: SW_DSA_OVER_LIMIT]

  If there is not enough disk space and *enforce_dsa=false*, generate an event.
  [SW_WARNING: SW_DSA_OVER_LIMIT]

How disk space analysis is implemented is undefined. However an implementation must account at least for the sizes of the files and control_files being copied.

Execution Phase

As the result of a *swcopy*, products and bundles, if specified, are copied to the target, which is a distribution software object.

When creating serial distributions, an implementation must support one or both of the POSIX.1 extended *cpio* or extended *tar* archive formats. Whether an implementation supports writing both archive formats or only one, and which of the formats is supported if only one, is implementation defined.

The relationship between the fileset loading and state transitions for *swcopy* is shown in the following list.

Copy each product:

1. Create the distribution catalog information for the product and its contained subproducts.
2. Copy each fileset in the product:
   a. Create the distribution catalog information for the fileset, setting the state to `transient`.
   b. Load the files for the fileset.
   c. Update the state of the fileset to `available`.

Copy each bundle:

1. Create the distribution catalog information for the bundle.

A depot update is defined as ‘‘copying or packaging a higher revision of a fileset than one that currently exists within a particular product revision in a depot’’.

When the new revision of the fileset has completed the copy, and the state of the fileset is set to *available*, also remove the catalog information for the existing fileset (or set the state of the existing fileset to *removed*), and remove any files that were contained in the old fileset but are not contained in the new fileset. See step 2c above.

For disk space analysis during a copy update, ensure that files that will be removed from a depot during update are accounted for, in addition to the normal analysis for new files and files being replaced.

- **File Loading**

  In this step, *swcopy* copies the files from the source onto the target.

  If a file load fails for any other reason such as a lost connection to the remote source or tape eject, then the fileset load fails. The following are the errors that can occur during the file loading step:

  — If an error or warning occurs while loading a file onto a target, an event is generated and, for errors, the implementation defined error handling procedures invoked.

  Whether remote files are loaded is implementation defined. If the file is on a remote file system generate an event if it was loaded.
  [SW_NOTE: SW_FILE_IS_REMOTE]

  Generate an event if it was not loaded.
  [SW_WARNING: SW_FILE_IS_REMOTE]

  — If the error was a source access problem, the user may attempt to correct the problem and start over with the fileset that had the failure.
  [SW_ERROR: SW_SOURCE_ACCESS_ERROR]

  Like install, the *reinstall_files* option, by default set to `true`, allows for administrator control of recopying up to date files independent of a recopy or an update. If set to `true`, files are copied independent of whether they are the same. If set to `false`, then they are checked first.

  Like *install*, the cksum check can take as long as actually transferring and recopying the file. The *reinstall_files_use_cksum* option, by default set to `true`, can be set to `false` to skip the checksum check.

  Copying a *sparse* fileset into a depot has the same behavior as copying a normal fileset into a depot. If the fileset has a different fileset revision but the same product revision as a fileset that is already in the depot, it replaces that fileset. If the fileset has a different product revision, then the copy task will result in both product revisions, and both filesets, residing in the depot.

- **Compression**

  If *compress_files=true* is specified, then the files must be compressed as follows in copying to the target distribution:

  — `INDEX` and `INFO` files must not be compressed

— All files that have the ***compression_state*** attribute undefined or its value set to `uncompressed` should be compressed. If the file cannot be compressed, generate an event:
[SW_ERROR: SW_COMPRESSION_FAILURE]

— All files that have the value of the ***compression_state*** attribute set to `not_compressible` must be copied without change.

— A source file which is already compressed, and which has the value of its ***compression_type*** attribute equal to the value of the ***compression_type*** extended option, should be copied without change.

— If a source file is already compressed, and the value of its ***compression_type*** attribute is different to the value of the ***compression_type*** extended option, the behavior is undefined. Unless the implementation can successfully uncompress the file and then compress it with the correct type, generate an event:
[SW_ERROR: SW_COMPRESSION_FAILURE]

If ***uncompress_files=true***, the files must be uncompressed as follows in copying to the target distribution:

— All files with a ***compression_state*** attribute value of `compressed` must be uncompressed as part of the copy. If the file cannot be uncompressed, generate an event:
[SW_ERROR: SW_COMPRESSION_FAILURE]

— All other files should be copied without change.

If neither ***compress_files=true*** nor ***2uncompress_files=true***", then the files are copied without change. Behavior when both are set to true is undefined.

- **Copying into Existing Products**

  When a fileset is copied into an existing target product, the attributes of this existing product may be affected as follows. If an attribute exists in the product from which the fileset came, the value of this attribute is set in the target product. Any attributes in the target product not found in the product from which the fileset came are left unaltered. It is possible for attributes to be set multiple times as filesets from different products are copied into the target product.

EXIT STATUS

See Chapter 3.

CONSEQUENCES OF ERRORS

See Chapter 3.

NAME

swinstall — install software

SYNOPSIS

```
swinstall [-p][-r][-c catalog][-f file][-s source][-t targetfile]
          [-x option=value][-X options_files][software_selections]
          [@ targets]
```

DESCRIPTION

The *swinstall* utility installs software from a distribution to installed_software objects on the targets. It may also configure the software. The software is not necessarily available for use until after it has been configured.

OPTIONS

The *swinstall* utility supports the following options. Where there is no description, the description in Chapter 3 applies.

`-c` ***catalog***

If this option is specified, then use the exported catalog structure at this path as the source of the `response` files.

If ***ask=true*** or ***ask=as_needed***, the control directories in the exported catalog structure are used for both the eventual source of the `response` files, and the control directory where the the `request` scripts are executed in order to create any needed `response` files.

`-f` ***file***

`-p`

`-r`

`-s` ***source***

`-t` ***targetfile***

`-x` ***option=value***

`-X` ***options_file***

OPERANDS

The *swinstall* utility supports the *software_selections* and *targets* operands described in Chapter 3.

The utility supports software selection operands with `[l=location]` part of the syntax, designating the *product location* directory that replaces the *product directory* attribute when installing the software.

EXTERNAL INFLUENCES

See Chapter 3 for descriptions of external influences common to all utilities.

Extended Options

The *swinstall* utility supports the following extended options. The description in Chapter 3 applies.

allow_downdate=false

allow_incompatible=false

ask=false

autoreboot=false

autorecover=false

autorecover_product

autoselect_dependencies=as_needed

autoselect_patches=true

defer_configure=false

defer_deleting_files=false

distribution_source_directory=implementation_defined_value

enforce_dependencies=true

enforce_locatable=true

enforce_scripts=true

enforce_dsa=true

installed_software_catalog=implementation_defined_value

logfile=implementation_defined_value

loglevel=1

match_target=false

patch_save_files=true

*patch_filter=**

patch_match_target=false

reinstall=false

reinstall_files=false

reinstall_files_use_cksum=true

save_modified_files=false

saved_files_directory=implementation_defined_value

select_local=true

software=

targets=

verbose=1

EXTERNAL EFFECTS

See Chapter 3.

EXTENDED DESCRIPTION

See Chapter 3 for general information. There are three key phases in the *swinstall* utility:

1. Selection phase
2. Analysis phase
3. Execution phase

Selection Phase

Multiple versions of a software product can exist from the source, distinguished by their respective ''version distinguishing attributes'' (*revision*, *architecture*, and *vendor_tag*). If the method described in Section 3.4.1 on page 38 results in an ambiguous selection, the following method is used to identify a single version:

> If *allow_incompatible=false*, the target *uname* attributes are used to filter the available products to only those that are compatible with the target systems, then the version with the highest possible *product revision* is chosen from this filtered list. If this filtering and selection of a highest revision does not result in a unique version, then no version is selected. If *allow_incompatible=true*, then only the highest revision is used to try to determine a unique version. In either case, if there is still an ambiguous selection, no version is selected. See Section 3.4.1.2 on page 42.

If a software selection has dependency specifications on other software, and the option *autoselect_dependencies=true*, the dependency software is attempted to be automatically selected using the same method to determine a single version. This automatically selected software is then installed along with the rest of the selected software. If *autoselect_dependencies=as_needed*, then dependency software is attempted to be automatically selected and installed only if the dependency is not already met on the target.

If a fileset has an exrequisite on another software object, and that other software object is part of the specified software selection, either explicitly or as part of another selection, then the fileset is excluded. If two filesets have exrequisites on each other, then the behavior is implementation defined.

If more than one ancestor is defined, then this new fileset is included in the selection list during update if either or both ancestors are currently installed and the *match_target* option is set to *true*.

Unless ancestor is explicitly defined, all filesets have the default ancestor of any revision less than itself, that is,

```
product_tag.fileset_tag,r<revision,a=architecture,v=vendor_tag).
```

For *swinstall*, each selection added to the selected software list must satisfy the following validation checks. If any of these checks result in an event with a status of [SW_ERROR], the selection is not added to the list and the implementation defined handling procedure can be invoked.

- If the selection is not available from the source, generate an event.
  [SW_ERROR: SW_SELECTION_NOT_FOUND]
- If a unique version can not be identified, generate an event.
  [SW_ERROR: SW_SELECTION_NOT_FOUND_AMBIG]
- If an attempt is made to select more than one version of a given product targeted for the same location, generate an event.
  [SW_ERROR: SW_EXISTING_MULTIPLE_VERSION]
- If *2allow_incompatible=true"*, for each target where the software selected is incompatible with that target (see Section 3.4.1.2 on page 42, generate an event.
  [SW_WARNING: SW_NOT_COMPATIBLE]

  If *allow_incompatible=false*, for each target where the software selected is incompatible with that target (see Section 3.4.1.2 on page 42, generate an event. The implementation defined error handling procedure is then invoked.
  [SW_ERROR: SW_NOT_COMPATIBLE]

- Check if a non-default product location has been specified. If ***enforce_locatable=true***, generate an event.
  [SW_ERROR: SW_NOT_LOCATABLE]

  If ***enforce_locatable=false***, generate an event.
  [SW_WARNING: SW_NOT_LOCATABLE]

- If the software is excluded, generate an event.
  [SW_NOTE: SW_EXREQUISITE_EXCLUDE]

If ***ask=true*** then execute the software `request` scripts for the selected software as described in the ***swask*** utility. See ***swask*** on page 78, extended description.

Selection of software to update can be done using the ***match_target*** option. The ***match_target*** option provides a shortcut for selecting software. By default set to `false`. When set to `true`, this option causes each fileset in the distribution to be included in the selection list if:

- A fileset with the same product and fileset tag exists in only one location on the system
- The fileset in the depot has an ***ancestor*** attribute that matches a single installed fileset
- It is the highest compatible version if more than one version in the depot matches, including accounting for superseding filesets

If there are multiple filesets that match (that is, multiple revisions are installed), it is not clear which if any should be updated, and an event is generated: [SW_WARNING: SELECTION_NOT_FOUND_AMBIG].

These selections are combined with any other selections specified in the other supported means (command line operands, a software_selections file or the ***software*** extended option), and then go through the standard software selection checks.

Analysis Phase

The target role uses the file size information and `checkinstall` scripts obtained from the source to determine whether or not the install utility proceeds on the given target. When failures occurs in the disk space analysis and `checkinstall` scripts, it is implementation defined whether or not to proceed with a partial list of software selections.

If any target generates an event with a status of [SW_ERROR] during any of the analysis operations, what software is attempted to be installed is determined by the implementation defined error handling procedures.

The target role checks the following requirements:

- If the target installed_software object does not exist on a host, create the path to the target with default attributes, and generate an event.
  [SW_NOTE: SW_SOC_CREATED]

- Check that selected filesets are not the same version as already installed. If ***2reinstall=false***", note that they will be skipped by generating an event.
  [SW_NOTE: SW_SAME_REVISION_SKIPPED]

  If ***reinstall=true***, note that they will be reinstalled by generating an event.
  [SW_NOTE: SW_SAME_REVISION_INSTALLED]

- Check that selected filesets are not lower versions of the fileset already installed on the host. If ***allow_downdate=false***, generate an event.
  [SW_ERROR: SW_HIGHER_REVISION_INSTALLED]

If *allow_downdate=true*, generate an event.
[SW_WARNING: SW_HIGHER_REVISION_INSTALLED]

- Execute vendor-supplied `checkinstall` scripts to perform product-specific checks of the target. If the `checkinstall` script returns an error, and *enforce_scripts=true*, generate an event and invoke the implementation defined error handling procedure.
[SW_ERROR: SW_CHECK_SCRIPT_ERROR]

  If the `checkinstall` script returns an error and *enforce_scripts=false*, generate an event.
[SW_WARNING: SW_CHECK_SCRIPT_ERROR]

  If the `checkinstall` script returns a warning, generate an event.
[SW_WARNING: SW_CHECK_SCRIPT_WARNING]

  If the script has a return code of 3, generate an event and unselect the fileset (or all filesets in the product for a product level script).
[SW_NOTE: SW_CHECK_SCRIPT_EXCLUDE]

- Verify that the needed dependencies of the filesets are met. If *enforce_dependenciesr= true*, generate an event.
[SW_ERROR: SW_DEPENDENCY_NOT_MET]

  If *enforce_dependencies=false*, generate an event.
[SW_WARNING: SW_DEPENDENCY_NOT_MET]

- Check that there is enough free disk space on the target file system to install the selected products. If there is not enough disk space and *enforce_dsa=true*, generate an event.
[SW_ERROR: SW_DSA_OVER_LIMIT]

  If there is not enough disk space and *enforce_dsa=false*, generate an event.
[SW_WARNING: SW_DSA_OVER_LIMIT]

- An implementation may generate the following events if disk space analysis encountered any problems that prevented the analysis. If *enforce_dsa=true*, generate an event.
[SW_ERROR: SW_DSA_FAILED_TO_RUN]

  If *enforce_dsa=false*, generate an event.
[SW_WARNING: SW_DSA_FAILED_TO_RUN]

How disk space analysis is implemented is undefined. However an implementation must account at least for the sizes of the files and control_files being installed, the additional sizes from the vendor supplied `space` file described in Section 5.2 on page 130, and the additional space required from saving files if *autorecover=true* and if required by the implementation defined recovery process.

Most revision checks during operation of install are done fileset by fileset. When checking for newer revisions of the fileset, the product revision is checked before the fileset revision:

- If one fileset's product has a revision higher than the other fileset's product revision, it is a newer revision.
- If the product's revisions are undefined or the same, then the fileset revisions are checked.

Execution Phase

The execution phase is the third part of the installation process, and is entered once either the selections have passed the analysis phase with no events with a status of [SW_ERROR] or if permitted by the implementation defined error handling procedures.

The relationship between the `preinstall` and `postinstall` scripts, fileset loading, and state transitions for *swinstall* is shown in the following list. Products are ordered by prerequisite dependencies if any. Fileset operations are also ordered by any prerequisites.

1. Install each product:
   a. Create the installed_software catalog information for the product and its contained subproducts.
   b. Run the `preinstall` script for the product.
   c. Install each fileset in the product:
      i. Create the installed_software catalog information for the fileset, setting the state to `transient` . Also update the state of any existing fileset that is being updated or downdated to `transient` .
      ii. Run the `preinstall` script for the fileset.
      iii. Load the files for the fileset.
      iv. Run the `postinstall` script for the fileset.
      v. Update the results of the scripts. Update the state of the fileset to `installed` . Also set the state of any existing fileset that is being updated or downdated to `removed` or remove the catalog information for that fileset.
   d. Run the `postinstall` script for the product.
   e. Once the catalog information for the last fileset in a particular product version has been removed due to update, like *swremove*, the catalog information for that product version should also be removed. See *swremove* on page 117.
2. Install each bundle:
   a. Create the installed_software catalog information for the bundle.
3. Configure each product (see ''executing configure scripts'' in *swconfig* on page 81.

   Configuration is done at this point by the *swinstall* utility only if *defer_configure=false*, the target directory is `/`, and no filesets with the *is_reboot* attribute equal to true have been installed.
   a. Run the `configure` script for the product.
   b. Configure each fileset in the product:
      i. Run the `configure` script for the fileset.
      ii. Update the result of the script. Update the state of the fileset to `configured` in the catalog for the installed_software object.

Configuration will not be executed by *swinstall* if the software creates a multiple version, the target directory is not `/`, or if the software is incompatible and *allow_incompatible=false* (see Section 3.4.1.2 on page 42). In these cases, *swconfig* may be used.

If events with a status of [SW_ERROR] are detected during the execution phase, the *swinstall* utility generates the appropriate event, any log entries, and invokes the implementation defined error handling procedures. For each fileset that failed, the installed_software catalog is updated to the state `corrupt`.

The *swinstall* utility will only remove catalog information for filesets being updated if they have the same product and fileset tag in the same location by default. By specifying a *supersedes* attribute, catalog information for filesets being updated that have changed names (have a different tag attribute) or are otherwise superseded by new functionality will be removed as well.

- **File Location**

  If an alternate root directory was specified (a value for *installed_software=path* other than `/`, then the alternate root directory is used as a prefix to the *file path* attribute to determine the file location in the file system. See Section 3.4.2 on page 42.

  The *file path* will be modified if the product is locatable and a new *product location* is specified (using the *l=location* software specification). The *product directory* part of the *file path* is replaced by the value *product location* attribute before a file is placed in the target file system.[12]

  If a *bundle location* is specified (using the `l=location` software specification when specifying a bundle), then the *bundle=location* will be prepended to the location specification for each `software_spec` in the contents of the bundle, prior to replacement of the *product directory* part of the *file path*.

- **Preinstall Scripts**

  In this step of the execution phase, *swinstall* executes product and fileset `preinstall` scripts.

  — If a `preinstall` script returns an error and *enforce_scripts=true*, generate an event and invoke the implementation defined error handling procedures.
  [SW_ERROR: SW_PRE_SCRIPT_ERROR]

  — If the `preinstall` script returns an error and *enforce_scripts=false*, generate an event.
  [SW_WARNING: SW_PRE_SCRIPT_ERROR]

  — If a `preinstall` script returns a warning, generate an event.
  [SW_WARNING: SW_PRE_SCRIPT_WARNING]

  Control scripts must adhere to the specifications in section Section 3.6.1 on page 56.

- **File Loading**

  In this step, *swinstall* loads the files from the source onto the target file system according to information obtained from the source distribution. All file types are created using the attributes defined for those files in the source distribution. Regular files (that is, those with a *file type* of `f`) are loaded using the content from the source distribution.

  If the source file is a regular file or a directory and its path already exists on the target file system as a symbolic link, then the symbolic link is followed and the file is stored in the path defined by the symbolic link.

---

12. If a product is locatable (has the *product=is_locatable* attribute set to `true`), all files that have the value of *product directory* as the initial part of their path will be installed to a new location if one has been specified. The *product directory* attribute is the base directory for the files that are locatable within a specific product.

If the source file is a symbolic link, then the existing path is replaced by symbolic link.

— If there are too many levels of symbolic links,[13] then the file is skipped and an event is generated.
[SW_WARNING: SW_FILE_WARNING]

The file owner and group names are set to the values specified for the *file owner* and *file gid* attributes for the source file. If the target host does not contain those file owner and group names, the file uid and gid are set to the numeric values specified for these attributes for the source file. If no values are specified for these attributes, the uid and gid are set to the effective uid and gid of the current process. See Section 5.2.14.3 on page 144.

— If the user or group of the file is not defined on the target host, or either of these attributes are not defined for the file, generate an event:
[SW_WARNING: SW_FILE_WARNING]

— If the *mode* attribute of the file has the set user id on execution (`S_ISUID`) bit set and either the *user* attribute of the file is not defined on the target host or the *user* attribute is not specified for the file, the corresponding mode bit in the file system will not be set when installing the file and an event is generated. See POSIX.1.
[SW_ERROR: SW_FILE_ERROR]

— If the *mode* attribute of the file has the set group id on execution (`S_ISGID`) bit set and either the *group* attribute of the file is not defined on the target host or the *group* attribute is not specified for the file, the corresponding mode bit in the file system will not be set when installing the file and an event is generated. See POSIX.1.
[SW_ERROR: SW_FILE_ERROR]

The value of the *file mode* attribute on the file is set to the value of the *file mode* attribute for the source file. An exception is that directories that already exist are not modified. If no values are specified for this attribute, the mode is set to the default file creation mode for the current process.

If there is an existing installed file that matches the values supplied in the distribution for the *path*, *cksum*, *date*, and *size* attributes, the file is not reloaded unless the user has specified that the fileset is being reinstalled.

The *reinstall* option is not sufficient for controlling whether files are reinstalled during update, reinstall or downdate, since it is only controls whether any of the same fileset at a target location is attempted or not, based on the catalog information stored on that target. The *reinstall_files* option, by default set to `false`, allows for the administrator to check for up-to-date files independent from the reinstall option. If set to `true`, files are installed independently of whether they are the same. If set to `false`, then the actual size, mtime and cksum atttributes of the installed files, as opposed to the catalog information about these files, are checked against the catalog information in the source before installing the file.

As it takes time to compute the cksum when the *reinstall_files* option is set to `false`, there is an option to skip that check that may be sufficient in secure or controlled environments. The *reinstall_files_use_cksum* option, by default set to `true`, can be set to `false` to skip the cksum check. In this case, only the size and mtime are checked when determining if the file is

13. It is not the intention of this Software Administration specification to define symbolic links in a manner inconsistent with POSIX.1. However, no approved POSIX standard currently contains symbolic links. This definition is a placeholder until such time as an approved standard provides the definition.

up to date.

If a file load fails for any other reason such as a lost connection to the remote source or tape eject, then the fileset install fails.

In order to aid cleanup of obsolete shared files, a new file type *x* is defined that directs the install execution phase to remove a file instead of installing it. Files of type *x* will have the paths resolved with respect to product *location* attributes just as normal files to determine the file to remove. If *defer_deleting_files=true*, then the files are not removed. Disk space analysis will account for files of this type by subtracting existing file sizes.

The following are problems that may occur during the file load step:

— If a problem occurs while loading a file onto a target, an event is generated and, for events with a status of [SW_ERROR], the implementation defined error handling procedures invoked. If there are too many levels of symbolic links, generate an event.
[SW_ERROR: SW_FILE_ERROR]

Whether remote files are installed is implementation defined. If the file is on a remote file system and was loaded, generate an event.
[SW_NOTE: SW_FILE_IS_REMOTE]

If it was not loaded, also generate an event.
[SW_WARNING: SW_FILE_IS_REMOTE]

— If a file can not be updated because it is busy, or it is a directory, then move that file to implementation defined location and generate an event. How these files are eventually removed is also implementation defined.
[SW_WARNING: SW_FILE_NOT_REMOVABLE]

— If the source becomes inaccessible for any reason during the process of loading files, generate an event.
[SW_ERROR: SW_SOURCE_ACCESS_ERROR]

If *autorecover=true*, then all files that are being updated are saved. It is implementation defined where these files are saved. The saved files for filesets in each product are removed in an implementation defined manner at some point after that product completes the execution phase.

Before overwriting any existing file, if the *save_modified_files* option is set to `true`, any file that has a size or cksum different to that in the installed software catalog will be saved to an implementation defined location. If the *save_modified_files* option is set to `false`, then these files may be overwritten.

After a product has been installed, files in the product of type *delete file* are removed if the *defer_deleting_files* option is set to `false`. If the *defer_deleting_files* option is set to `true`, then files of this type are not deleted.

- **Compression**

When installing files, all compressed files are uncompressed as follows as part of file loading:

— All files that have the *compression_state* attribute value of `compressed` are uncompressed, according to the value of *compression_type* attribute. The way in which this is done is implementation defined. If the file cannot be uncompressed, generate an event.
[SW_ERROR: SW_COMPRESSION_FAILURE]

- **Postinstall Scripts**

In this step, *swinstall* executes the product and fileset `postinstall` scripts.

— If a `postinstall` script returns an error and *enforce_scripts=true*, generate an event and invoke the implementation defined error handling procedures.
[SW_ERROR: SW_POST_SCRIPT_ERROR]

— If the `postinstall` script returns an error and *enforce_scripts=false*, generate an event.
[SW_WARNING: SW_POST_SCRIPT_ERROR]

— If a `postinstall` script returns a warning, generate an event.
[SW_WARNING: SW_POST_SCRIPT_WARNING]

- **Kernel Scripts** Special customization and install steps are executed when processing kernel filesets. Kernel filesets are those for which the value of the *is_kernel* attribute is `true`, causing *swinstall* to modify the fileset load order and to invoke the *postkernel* script. Apart from this, *swinstall* has no special functionality for installing kernels.

The *postkernel* scripts are those specified by the value of the *product=postkernel* attribute, or by the implementation defined default. The functions invoked by this *postkernel* script are implementation defined. Examples of use include rebuilding the kernel or moving a new default kernel into place.

The *postkernel* script is not interactive, and issues all informative and error messages to stdout and stderr, which redirects it to the log file. In addition, the *postkernel* script provides a standard return value indicating success (0, that is, zero), error (1) or warning (2).

— If the *postkernel* script has an error, an event is generated and the implementation defined error handling procedures are invoked.
[SW_ERROR: SW_POSTKERNEL_ERROR]

— If the *postkernel* script has an warning, an event is generated.
[SW_WARNING: SW_POSTKERNEL_WARNING]

The kernel filesets are processed before the rest of the filesets. All products are first processed for their kernel filesets, and then all products are processed for their non-kernel filesets. The ordering of products and filesets also adheres to prerequisites, just as normal filesets:

1. Install the kernel filesets for each product:
    a. Create the installed_software catalog information for the product.
    b. Run the `preinstall` script for the product.
    c. Install each kernel fileset in the product:
        i. Create the installed_software catalog information for the fileset, setting the state to `transient`.
        ii. Run the `preinstall` script for the kernel fileset.
        iii. Load the files for the kernel fileset.
        iv. Run the `postinstall` script for the kernel fileset.
        v. Update the results of the scripts. Update the state of the fileset to `installed`.
    d. Run the `postinstall` script for the product.
2. Perform steps after installing kernel filesets by calling the zero or more scripts defined by the *product.postkernel* attributes of each product with a kernel fileset, and the

implementation defined default *postkernel* script if a product does not define a *product.postkernel* attribute.

3. Install the rest of the filesets for each product as described in the execution phase description for *swinstall*, omitting the kernel filesets already installed.

4. After all filesets have been installed, the implementation defined reboot procedure is executed on the target host if a fileset with the is_reboot attribute set to true has been installed and the **SW_ROOT_DIRECTORY** is `/`, and *autoreboot=true*. If rebooting, the software is not configured. The products will be configured after the reboot in an implementation defined manner using the *swconfig* utility.

5. If not rebooting, then configure each product as described in the execution phase for *swinstall* (including both kernel and non-kernel filesets).

- **Rebooting the System**

  If this step is required, the target role executes the implementation defined reboot procedure after all products have been installed. It is performed only when software is installed that requires a reboot as part of its installation (indicated by the *is_reboot* fileset attribute).

  If the system fails to execute the reboot step, generate an event.
  [SW_ERROR: SW_CANNOT_INITIATE_REBOOT]

- **Recovery**

  Within the execution phase of a particular product (from the product preinstall step through the product postinstall step), if any `preinstall` script, file loading, or `postinstall` script fails for a fileset, that fileset is deemed to have failed during install. The failure of a product `postinstall` script is considered the same as if all fileset `postinstall` scripts had failed.

  If such a failure occurs and *autorecover=false*, no recovery is provided for any filesets deemed to have failed during install, and the fileset *state* attribute of those filesets is set to `corrupt`. No further attempt is made to install such filesets during the current invocation of *swinstall*. Install can proceed on other filesets which did not fail during install.

  If an install failure occurs and *autorecover=true*, at least the following minimal error recovery must be provided at the fileset level. Additional recovery behavior, such as recovering the whole product or all products, is implementation defined. Additionally, if *enforce_dependencies=true*, implementations should take into account other filesets in the product that have a dependency on that failed fileset.

  The recovery is initiated at the point of failure, recovering the affected filesets, then continuing from the point of failure to the remaining filesets.

  Recovery involves running `unpostinstall` scripts, restoring files, and running `unpreinstall` scripts. The relationship between these steps for each product is shown in the following list.[14]

  1. Create the installed_software catalog information for the product.

  2. Run the `preinstall` script for the product.

14. In general, the order when *autorecover=true* is the same as normally done for successful steps, and the reverse when recovery steps are being executed.

If the `preinstall` script fails, or if all filesets have failed, run the product `unpreinstall` script, remove the catalog information for the product and go on to the next product.

3. Install each fileset in the product

   a. Create the installed_software catalog information for the fileset, setting the state of it and the fileset being updated to `transient`.

   b. Run the `preinstall` script for the fileset.

      If the `preinstall` script fails, run the `unpreinstall` script for the fileset, remove the catalog information for the fileset, restore the state of the fileset being updated, and go on to the next fileset.

   c. Load the files for the fileset.

      Before loading any files, save any existing files that will be overwritten by a file being loaded from the fileset, and then load the files for the fileset. If the fileset loading fails, restore the saved files for the fileset, delete all loaded files for which there is no saved file, and perform the previously described recovery step for this fileset.

   d. Run the `postinstall` script for the fileset.

      If the `postinstall` script fails, run the `unpostinstall` script for the fileset, and perform steps (3b) and (3c) for this fileset.

   e. Update the results of the scripts. Update the state of the fileset to `installed`.

4. Run the product `postinstall` script.

   If the product `postinstall` script fails, run the product `unpostinstall` script, and perform each of the previously described recovery steps for each fileset.

5. This is the first point in the process where the saved files may be removed. Remove the catalog information for filesets that were updated, or set the state of those filesets to `removed`.

   Once the catalog information for the last fileset in a particular product version has been removed due to update, like *swremove*, the catalog information for that product version should also be removed. See *swremove* on page 117.

   During update, it may be useful to have either the whole product or no part of it installed if there is a failure in any fileset or product control script of file loading. Set the *autorecover_product* option to `true` to extend the *autorecover* behavior. The behavior is identical except that any failure unwinds the entire product back to where it was before the install. In the case of a product *postinstall* script failure, the behavior is the same. From an implementation standpoint, all files in the product that are being updated are saved until the product completes successfully.

- **Sparse Updates**

  Updates can be packaged as sparse updates. When selecting a sparse fileset, swinstall has the following behavior:

  — Unless explicitly specified, swinstall will choose the highest compatible revision of a product and fileset whether it is sparse or not.

  — Installing a sparse fileset follows the same rules for update, downdate and reinstall as other filesets.

— If the ancestor of the sparse fileset is not already installed, and it is available from the depot, it is included in the list of software to install. If the ancestor is not already installed, and is not in the depot, then the sparse fileset can not be installed and an event is generated: [SW_ERROR: SW_DEPENDENCY_NOT_MET].

— When installing a sparse fileset, the existing fileset's catalog information is included as the catalog information of the sparse fileset, and then augmented with the sparse filesets file contents (adding and possibly deleting files). Then, as with normal update, the previous fileset's catalog information is deleted.

- **Patches**

Installation of patch products will, in general, follow the same rules as software installation. The key difference will be that a filtering mechanism is provided that allows only patches meeting specified criteria selected. The additional filtering mechanism will be provided through support of *category* in software specifications, and *autoselect_patches*, *patch_match_target* and *patch_filter* options.

When a patch is installed, the fileset that has been patched has the *applied_patches* attribute updated to include that patch, and the contained files information updated to include the patched file attributes.

When a patch is installed, by default it has the *patch_state* of `applied`. When that patch is committed, or it has been installed without saving roll-back files, it has the state of `committed`. When that patch is superseded, the *patch_state* is set to `superseded`, and the *superseded_by* attribute is set to the software_spec of the superseding fileset.

If a fileset is selected for installation and patch filesets for that *base* fileset exist in the same source depot, all applicable patches will by default be selected if the *autoselect_patches* option is set to `true`. The following additional rules will apply:

— Patch software selections will be filtered as defined by the *patch_filter* option.

— If more than one patch to a base fileset exists, patches to that fileset will be examined to determine if any patches have been superseded by later versions. The superseding patch will be installed and superseded patches will not be selected.

Patches can also be explicitly specified.

Patches that are installed in a separate session after the base product has been installed, can be selected explicitly or by matching the installed software using the *patch_match_target* option. If patches are selected via matching, the superseding mechanism described above will be used to determine the most recent patches to a fileset.

Consistent with the current level of expression support in the POSIX standard software specifications, a version qualifier can be repeated (for "AND" criteria), and the pipe symbol ("|") can be used within qualifiers (for "OR" criteria).

Patches can be explicitly installed without autoselection or matching the target by specifying one or more *software_spec* operands.

In order to accommodate patching of libraries (for example `libc.a`), a new file type of *a* is used. When loading a file of type *a* (archive file), *swinstall* temporarily installs the .o file to the target path specified, integrates it into the archive specified by the *archive_path* attribute of the file, and then removes the .o file. If rollback is enabled (see below), the original .o file is automatically extracted first and saved so that it can be replaced. Disk Space Analysis is performed as needed to account for these operations.

If patch filesets and normal filesets are being installed in the same session, then each patch fileset is considered to have an implied prerequisite on the fileset that it is patching with respect to ordering. In other words, the product containing the patch fileset will be installed (or copied into serial distributions) after the one or more products containing that patches *ancestors*.

If an *ancestor* fileset has the *is_kernel* attribute set to `true` then the fileset patching it must have the *is_kernel* attribute set to `true` in order for it to be installed in the kernel phase of the execution. Otherwise, the patch will be installed along with other non-kernel filesets.

Allowing later rollback of a patch requires saving files via the *patch_save_files* option. This option is set to `true` by default, which results in files replaced by the patch being saved for later rollback. The directory where the saved files are stored is specified by the *saved_files_directory* option. The structure of the rollback directory is undefined.

Installation of a new version of a base fileset will result in removal of all filesets which patch the base fileset, together with any files saved for potential roll-back.

EXIT STATUS

See Chapter 3.

CONSEQUENCES OF ERRORS

See Chapter 3.

NAME

swlist — list software catalog

SYNOPSIS

```
swlist [-d][-r][-v][-a attribute][-c catalog][-f file][-l level]
       [-t targetfile][-x option=value][-X options_file]
       [software_selections][@ targets]
```

DESCRIPTION

The *swlist* utility displays information about software that has been installed on a system or is in a distribution.

When combined with *swmodify* there is a complete read/write interface to the installed_software and distribution catalog information.

OPTIONS

The *swlist* utility supports the following options. Where there is no description, the description in Chapter 3 applies.

`-a` ***attribute***

Specifies which attributes to list.

Multiple attributes may be listed by specifying multiple `-a` ***attribute*** options. Only attributes that apply to each object listed are included for that object. When used with the `v` option, the attributes are in the software definition file format. When the `v` option is not specified then the listing format is undefined (see *one_liner* extended option).

In addition to all attribute names defined in Software Administration specification, three additional items are supported by the `-a` *attribute* option:

create_date

If this value is specified, *swlist* returns a sequence of characters representing the date associated with the *create_time* attribute.

The format of this sequence of characters in the POSIX Locale is equivalent to the default *date* format described in POSIX.2.

```
date "+%a %b %e %H:%M:%S %Z %Y"
```

The format for other Locales is undefined.

mod_date

If this value is specified, *swlist* returns a sequence of characters representing the date associated with the *mod_time* attribute.

The format of this sequence of characters in the POSIX Locale is equivalent to the default *date* format described in POSIX.2.

```
date "+%a %b %e %H:%M:%S %Z %Y"
```

The format for other Locales is undefined.

software_spec

If this value is specified, *swlist* returns the conformant `software_spec` for the object, as defined in Section 3.4.1.2 on page 42, instead of listing the identified objects.

The `software_spec` includes the *tag* of the object, the *tag* of the associated product (if this object is a fileset or subproduct), and the version distinguishing attributes of this object or its associated product (if this object is a fileset or subproduct).

These additional items can also be used with the *one_liner* extended option.

`-c` *catalog*
Provides a means to list the full catalog structure.

If the `-c` option is specified, output from *swlist* is written to an exported catalog structure instead of stdout.

The `-c` option specifies a directory below which the catalog information for the specified objects and attributes are stored. The exported catalog structure is used both for distributions and installed_software catalog information. See Section 5.1.1 on page 126.

`-d`

`-f` *file*

`-l` *level*
Specifies level at which to list the objects below the specified software.

Option *level* may have values from the enumerated list:

```
bundle
product
subproduct
fileset
control_file
file
category
```

If the `-l` *level* option is not included, then only the object at the level directly below the specified software or software_collection is listed. See Table 4-1.

Software Selection	Level Listed
none specified	products
bundle	products
product	filesets
subproduct	filesets
fileset	files

Table 4-1 Default Levels

If no level is specified for bundle and subproduct specifications, all the available or currently installed product and fileset objects, resolved recursively, are listed.

Multiple `-l` *level* options may be used to explicitly control what objects are included.

The categories of available patches can be listed for patches that have included category objects in their definition. These categories can be listed using the `-l` *category* specification.

`-r`

`-t` *targetfile*

`-v` List all the attribute value pairs of the objects specified.

The `-v` option specifies that the format of the output is in the `INDEX` file format, as defined in Section 5.2 on page 130. Which attributes and objects are included is controlled by other options and operands. If the `-a` option is defined, then only those attributes are listed, otherwise all attributes are listed. If there is no `-v` option, then listing format is undefined (see *one_liner* extended option).

-x *option=value*

-X *options_file*

OPERANDS

The *swlist* utility supports the *software_selections* and *targets* operands described in Chapter 3. If no *software_selections* are provided, all software in the catalog (either distribution or installed software) is selected.

EXTERNAL INFLUENCES

See Chapter 3 for descriptions of external influences common to all utilities.

Extended Options

The *swlist* utility supports the following extended options. The description in Chapter 3 applies.

distribution_target_directory=implementation_defined_value

installed_software_catalog=implementation_defined_value

one_liner=implementation_defined_value

select_local=true

software=

targets=

EXTERNAL EFFECTS

See Chapter 3 for general information.

EXTENDED DESCRIPTION

See Chapter 3 for general information. There are two phases in the *swlist* utility:

1. Selection phase
2. Execution phase

Selection Phase

If there are no software selections specified, then all software from the catalog is processed. Otherwise, each selection added to the selected software list must satisfy the following validation check.

- If the selection is not available from the catalog file, generate an event.
  [SW_ERROR: SW_SELECTION_NOT_FOUND]

Unlike all other commands, *swlist* includes all software that matches a specification, even if the specification is ambiguous.

Implementations that support the *removed* state need to address *swlist* just as they do for source depots with *swinstall.* These implementations should at least document the behavior for *swlist* and other operations on *removed* filesets. In particular, for *swlist*, if attempting to list a fileset from the source that has a *removed* state, the behavior should be the same as if it was not there, generating an event.
[SW_ERROR: SW_SELECTION_NOT_FOUND]

When listing a *sparse* fileset in a depot or in installed software, there are no differences when filesets is sparse or normal.

Execution Phase

The attributes for the selections determined from the previous phase are listed in the formats defined by the options.

EXIT STATUS

See Chapter 3.

CONSEQUENCES OF ERRORS

See Chapter 3.

NAME

swmodify — modify software catalog

SYNOPSIS

```
swmodify [-d][-r][-p][-u][-a attribute=value][-c catalog][-f file]
         [-t targetfile[-x option=value][-X options_file]
         [software_selections][@ targets]
```

DESCRIPTION

The *swmodify* utility provides an object and attribute update, create, and delete interface to the distribution and installed_software catalog information independent of the other utilities. When combined with *swlist*, there is a complete read/write interface to the installed_software and distribution catalog information.

OPTIONS

The *swmodify* utility supports the following options. Where there is no description, the description in Chapter 3 applies.

`-a` ***attribute=value***

As an alternative to using a software definition file format to describe the file attributes, this option may be used to add or modify a single attribute (e.g. ***is_locatable***). If combined with the `-u` option, this may be used to delete an attribute.

Only one of the `-c` ***catalog*** and `-a` ***attribute*** options may be specified.

`-c` ***catalog***

This option specifies the pathname of the catalog information. If it is a file, then it will be a file using the software definition file syntax, in Section 5.2 on page 130 that defines the objects and attributes desired to be created or modified.

If it is a directory, then it will have the exported catalog structure. For example, this could be a directory containing the output of the `swlist -c` command.

Only one of the `-c` ***catalog*** and `-a` ***attribute*** options may be specified.

`-d`

`-f` ***file***

`-p`

`-r`

`-t` ***targetfile***

`-u`

Deletes the objects or attributes specified.

`-x` ***option=value***

`-X` ***option_file***

OPERANDS

The *swmodify* utility supports the *software_selections* and *targets* operands described in Chapter 3.

This utility need not support a target distribution in the serial format.

EXTERNAL INFLUENCES

See Chapter 3 for descriptions of external influences common to all utilities.

Extended Options

The *swmodify* utility supports the following extended options. The description in Chapter 3 applies.

distribution_target_directory=implementation_defined_value

installed_software_catalog=implementation_defined_value

files=

logfile=implementation_defined_value

loglevel=1

patch_commit= false

select_local=true

software=

targets=

verbose=1

Standard Input

Input Files

The source input files may be in one of the following:

- Software definition file, described in Section 5.2 on page 130.
- Exported catalog structure, described in Section 5.2 on page 130.

  Note that this structure may be used to describe the installed_software catalog information. There is a separate *product.instance_id* for each version of the product.[15]

EXTERNAL EFFECTS

See Chapter 3.

EXTENDED DESCRIPTION

See Chapter 3 for general information. The *swmodify* utility consists of three phases:

1. Selection Phase
2. Analysis Phase
3. Execution Phase

Selection Phase

- **Specifying the Source**

  The source selection differs from the general information in Chapter 3 in that the source is a catalog file, or set of catalog files in the software packaging layout format instead of a distribution, so there is no access control events for accessing the catalog file.

15. An installed version is distinguished by the same attributes as in a distribution, plus the *location* attribute.

If the file parsing discovers syntax errors, or missing but required attributes, then generate an event.
[SW_ERROR: SW_SOURCE_ACCESS_ERROR]

- **Software Selections**

  If there are no software selections specified, then all software from the catalog is processed. Otherwise, each selection added to the selected software list must satisfy the following validation checks. If any of these checks result in an error, the selection is not added to the list.

  — If the selection is not available from the catalog file, generate an event.
  [SW_ERROR: SW_SELECTION_NOT_FOUND]

  — If a unique version can not be identified, generate an event.
  [SW_ERROR: SW_SELECTION_NOT_FOUND_AMBIG]

  Analysis Phase

  See Chapter 3.

  Execution Phase

  The execution phase modifies the target catalog. Certain errors can occur when modifying the catalog:

  — If a file can not be found in order to look up its attributes for modifying the catalog, then generate an event.
  [SW_ERROR: SW_FILE_NOT_FOUND]

  — More complex rules apply when modifying attributes that inherit from the product to the fileset level. The *filesets* and *is_locatable* attributes are updated only by *swpackage* and *swmodify*. If a fileset definition is removed with *swmodify,* the *filesets* attribute is updated.

  — If *swmodify* is used to change a fileset *is_locatable* attribute, then the corresponding product attribute is recalculated.

- **Patches**

  To recover disc space, users may wish to remove versions of files saved by installation of patches after they are comfortable with the operation of a patch. To commit a patch (remove the rollback files), *swmodify* is invoked on the patch by setting the option *patch_commit=true*.

EXIT STATUS

See Chapter 3.

CONSEQUENCES OF ERRORS

See Chapter 3.

NAME

swpackage — package distribution

SYNOPSIS

```
swpackage [-p][-f file][-s psf][-x option=value][-X options_file]
          [software_selections][@ targets]
```

DESCRIPTION

The *swpackage* utility packages files from the local host into software objects that can be managed by the utilities in this Software Administration specification using the definitions from a PSF. The *swpackage* utility packages software into distributions that can be installed, copied or otherwise distributed or managed.

OPTIONS

The *swpackage* utility supports the following options. Where there is no description, the description in Chapter 3 applies.

`-f` *file*

`-p`

`-s` *psf*

This option specifies the pathname of the PSF which describes the details of the packages that *swpackage* operates on.

`-x` *option=value*

`-X` *options_file*

OPERANDS

The *swpackage* utility supports the *software_selections* and *targets* operands described in Chapter 3 with one exception. The utility may support only a single, local distribution target.

If no *software_selections* are provided, all software described by the PSF is selected.

Whether data on an existing target distribution in serial format is overwritten or merged is implementation defined.

EXTERNAL INFLUENCES

See Chapter 3 for descriptions of external influences common to all utilities.

Extended Options

The *swpackage* utility supports the following extended options. The description in Chapter 3 applies.

distribution_target_directory=implementation_defined_value

distribution_target_serial=implementation_defined_value

enforce_dsa=true

follow_symlinks=false

logfile=implementation_defined_value

loglevel=1

media_capacity=0

media_type=directory

psf_source_file=psf

reinstall_files=false

reinstall_files_use_cksum=true

software=

verbose=1

Product Specification File

See Chapter 5.

EXTERNAL EFFECTS

See Chapter 3.

EXTENDED DESCRIPTION

The *swpackage* utility consists of three phases:

1. Selection Phase
2. Analysis Phase
3. Execution Phase

Selection Phase

- **Specifying Targets**

  The target selection differs from the general information in Chapter 3, in that there may be only one target. If the target is a serial distribution, *swpackage* sets default tape types and sizes as described in the ‘‘extended options’’ description in this man-page definition.

- **Specifying the Source**

  The source selection differs from the general information in section Chapter 3, in that the source is a PSF, instead of a distribution. Hence there are no access control events for accessing the PSF.

  The selection phase reads (and parses) the PSF to obtain the information from the source PSF.

  — The product, subproduct, and fileset structure

  — The files contained in each fileset

  — The attributes associated with these objects.
  If the file parsing discovers syntax errors, or missing but required attributes, generate an event.
  [SW_ERROR: SW_SOURCE_ACCESS_ERROR]

- **Software Selections**

  If there are no software selections specified, then all software from the PSF is processed. Otherwise, each selection added to the selected software list must satisfy the following validation checks. If any of these checks result in an error, the selection is not added to the list.

  — If the selection is not available from the PSF, generate an event.
  [SW_ERROR: SW_SELECTION_NOT_FOUND]

  — if a unique version can not be identified, generate an event.
  [SW_ERROR: SW_SELECTION_NOT_FOUND_AMBIG]

Analysis Phase

The package analysis phase follows the following steps:

- Checks the dependency specifications for irregularities (such as circular prerequisites or missing dependencies).
(SW_WARNING: SW_DEPENDENCY_NOT_MET)

- Before a new storage directory is created, *swpackage* checks to see if this product version has the same identifying attributes as an existing product version, namely the same tag, revision, architecture, and vendor_tag. If all the identifying attributes match, then the user is repackaging (modifying) an existing version. Note if a fileset within that product is being repackaged by generating an event.
[SW_NOTE: SW_SAME_REVISION_INSTALLED]

- Checks existence and attributes of the control_files and files that the PSF defines. If any are missing, generate an event.
[SW_ERROR: SW_FILE_NOT_FOUND]

- Check that there is enough free disk space on the target file system to package the selected products. If *enforce_dsa=true*, generate an event.
[SW_ERROR: SW_DSA_OVER_LIMIT]

  If *enforce_dsa=false*, generate an event.
  [SW_WARNING: SW_DSA_OVER_LIMIT]

Execution Phase

The execution phase packages the source files and information into a product, and create/merge the product into the target distribution.

When creating a serial distribution, an implementation must support one or both of POSIX.1 extended **cpio** or extended **tar** archive formats. Whether an implementation supports writing both archive formats or only one, and which format is supported if only one, is implementation defined.

When packaging a product, the storage directory within the target distribution is created/updated directly by *swpackage*. For each unique version of the product, a directory is created using the defined *product.tag* attribute and a unique sequence number for all the product versions which use the same tag as specified in Chapter 5.

The *swpackage* command generates certain attributes as specified in Section 5.2 on page 130.

More complex rules apply when packaging attributes that inherit from the product to the fileset level. The *filesets* and *is_locatable* attributes are updated only by *swpackage* and *swmodify*. When packaging, the value of the *filesets* attribute is set to include all the filesets in the PSF, plus any others that exist in the distribution already but are not in the PSF. In the latter case, the user is warned that the PSF is not complete.

If undefined, the *product.is_locatable* attribute is set by *swpackage* if any of the filesets in the *filesets* list are locatable. If none of the filesets are locatable, or if that cannot be determined, then the value of the *product.is_locatable* attribute is set to `false` . If defined, the *product.is_locatable* attribute is used to define the value of the *is_locatable* attribute for any filesets that do not have *is_locatable* defined. If the value of the *is_locatable* attribute is defined at both the product and fileset level, then the fileset definition overrides the product definition.

Certain errors can occur when packaging the files:

- If a file can not be added to the distribution for any reason, generate an event.
  [SW_ERROR: SW_FILE_ERROR]

Just as is done for *swcopy*, *swpackage* will support the *reinstall_files* and *reinstall_files_use_cksum* options. The semantics are the same.

When packaging a *sparse* fileset, only the files contained within the sparse fileset are included in the package definition. If an ancestor is not defined, then this sparse fileset can be applied to any fileset with the same tag, product tag, architecture, vendor_tag, and with a lower revision.

EXIT STATUS

The *swpackage* utility returns:

0 The products specified in the PSF were successfully packaged onto the media.

1 An error occurred in parsing the PSF. The media was not modified.

2 An error during the packaging operation. The media has been modified. Review the log file for details.

CONSEQUENCES OF ERRORS

See Chapter 3.

NAME

swremove — remove software

SYNOPSIS

```
swremove [-d][-r][-p][-f file][-t targetfile][-x option=value]
         [-X options_file][software_selections][@ targets]
```

DESCRIPTION

The *swremove* utility performs the opposite function of the install software utility or the copy software utility. It removes installed software or software stored in a distribution.

The *swremove* utility removes software installed at the local host or at the targets specified on the command line. It also removes software from local or remote distributions.

OPTIONS

The *swremove* utility supports the following options. Where there is no description, the description in Chapter 3 applies.

`-d`

`-f` *file*

`-p`

`-r`

`-t` *targetfile*

`-x` *option=value*

`-X` *options_file*

OPERANDS

The *swremove* utility supports the *software_selections* and *targets* operands described in Chapter 3.

This utility need not support a target distribution in the serial format.

EXTERNAL INFLUENCES

See Chapter 3 for descriptions of external influences common to all utilities.

Extended Options

The *swremove* utility supports the following extended options. The description in Chapter 3 applies.

autoselect_dependents=false

distribution_target_directory=implementation_defined_value

enforce_dependencies=true

enforce_scripts=true

installed_software_catalog=implementation_defined_value

logfile=implementation_defined_value

loglevel=1

select_local=true

software=

targets=

verbose=1

EXTERNAL EFFECTS

See Chapter 3.

EXTENDED DESCRIPTION

See Chapter 3 for general information. The *swremove* utility consists of three main phases.

1. Selection phase
2. Analysis phase
3. Execution phase.

Selection Phase

As opposed to *swinstall*, software selections apply to the target distribution or installed_software.

Each specified selection must pass the following checks. If a specification does not pass a check, the implementation defined error handling procedure is invoked.

- If the selection is not found, generate an event.
  [SW_WARNING: SW_SELECTION_NOT_FOUND]
- If the selection is not found at that product directory, generate an event.
  [SW_WARNING: SW_SELECTION_NOT_FOUND_RELATED]
- If a single version of the software is not uniquely identified from the product and product attributes specified, generate an event.
  [SW_ERROR: SW_SELECTION_NOT_FOUND_AMBIG]

Add any dependent software to the selection list if ***autoselect_dependents=true***.

Analysis Phase

This section details the analysis phase. The analysis phase occurs before the removing of files begins, and involves executing checks to determine whether or not the removal should be attempted. No aspect of the target host environment is modified, so canceling the removal after these operations has no negative effect.

The target role makes the following checks:

- When removing installed software, execute vendor-supplied `checkremove` scripts to perform product-specific checks of the target. If ***enforce_scripts=true*** and the `checkremove` script returns an error, an event is generated and the implementation defined error handling procedure is invoked.
  [SW_ERROR: SW_CHECK_SCRIPT_ERROR]

  If the `checkremove` script returns an error and ***enforce_scripts=false***, generate an event.
  [SW_WARNING: SW_CHECK_SCRIPT_ERROR]

  If the script had a warning, generate an event.
  [SW_WARNING: SW_CHECK_SCRIPT_WARNING]

  If the script has a return code of 3, generate an event and unselect the fileset (or all filesets in the product for a product level script).
  [SW_NOTE: SW_CHECK_SCRIPT_EXCLUDE]
- Verify that the dependencies are met. The *swremove* utility does not remove a fileset if it is required by other filesets that have not been selected for removal or cannot be removed.

When *enforce_dependencies=true* and a non-selected fileset depends on a selected fileset, an event is generated and the implementation defined error handling procedure is invoked.
[SW_ERROR: SW_DEPENDENCY_NOT_MET]

When *enforce_dependencies=false*, generate an event.
[SW_WARNING: SW_DEPENDENCY_NOT_MET]

If a software object has been specified for removal, and there is a bundle referring to that object that has not also been specified for removal, the behavior is implementation defined. Likewise, if a fileset or subproduct object has been specified for removal, and there is a subproduct referring to that object that has not also been specified for removal, the behavior is implementation defined.

Execution Phase

For installed_software, the sequential relationship between the `unconfigure`, `preremove`, and `postremove` scripts, and removing files for *swremove* is shown in the following list. The `unconfigure` scripts are only run if the target directory is `/`.

1. Unconfigure each product:

   If the fileset has been configured more than once, the `unconfigure` script must unconfigure each instance.

   a. Unconfigure each fileset in the product:

      i. Run the `unconfigure` script for the fileset.

      ii. Update the result of the script. Update the state of the fileset in the product to `installed` in the database for the installed_software object.

   b. Run the `unconfigure` script for the product.

2. Remove each product:

   a. Run the `preremove` script for the product.

   b. Remove each fileset in the product:

      i. Update the state of the fileset to `transient` in the catalog for the installed_software object.

      ii. Run the `preremove` script for the fileset.

      iii. Remove the files for the fileset.

      iv. Run the `postremove` script for the fileset.

      v. Update the results of the scripts. Update the state of the fileset to `removed` in the catalog for the installed_software object or remove the catalog information for the fileset.

   c. Run the `postremove` script for the product.

   d. If the catalog information has been removed for all filesets in the product, an implementation can also remove the catalog information for the product and its contained subproducts.

3. Remove each bundle:

   a. Remove the installed_software catalog information for the bundle.

4. If the catalog information has been removed for all products and bundles in the installed_software object, an implementation can also remove the catalog information for the installed_software object.

For each fileset that failed to be removed, the installed_software catalog information is updated to the state `corrupt`.

- **Executing Pre-remove Scripts**

  In this step of the execution phase, *swremove* executes the software `preremove` scripts.

  — If a `preremove` script returns an error and *2enforce_scripts=true"*, generate an event and invoke the implementation defined error handling procedures.
  [SW_ERROR: SW_PRE_SCRIPT_ERROR]

  — If the `preremove` script returns an error and *enforce_scripts=false*, generate an event.
  [SW_WARNING: SW_PRE_SCRIPT_ERROR]

  — If a `preremove` script returns an warning, generate an event.
  [SW_WARNING: SW_PRE_SCRIPT_WARNING]

  When *swremove* is removing from a distribution, no scripts must be run.

- **File Removing**

  In this step, *swremove* removes the files from the target. The target role attempts to remove each file from the target file system according to information obtained in the software_selections sent.

  If *swremove* cannot remove a file (either because the file is busy [ETXTBSY], or for some other reason), the file name and the reason are logged so an administrator can take corrective action.
  [SW_WARNING: SW_FILE_NOT_REMOVABLE]

  If a filename is a symbolic link, the target is not removed. To achieve this behavior, the *swremove* utility handles symbolic links according to these rules:

  — If a file was recorded in the catalog as a symbolic link to another file, and it is still a symbolic link on the file system, remove the symbolic link, but do not remove the target file.

  — If a file was recorded in the catalog as a file, but exists as a symbolic link on the file system, remove the symbolic link, but do not remove the target file.

  — If the pathname to the file includes a symbolic link, this path is followed and the correct file is removed.

  All files that are targets of symbolic links are removed when the fileset to which they belong is removed.

- **Executing Post-remove Scripts**

  In this step of the execution phase, *swremove* executes software `postremove` scripts.

  — If a `postremove` script returns an error and *enforce_scripts=true*, generate an event and invoke the implementation defined error handling procedures.
  [SW_ERROR: SW_POST_SCRIPT_ERROR]

  — If the `postremove` script returns an error and *enforce_scripts=false*, generate an event.
  [SW_WARNING: SW_POST_SCRIPT_ERROR]

— If a `postremove` script returns an warning, generate an event.
[SW_WARNING: SW_POST_SCRIPT_WARNING]

- **Kernel Reconfiguration**

  If the *is_kernel* attribute of the fileset is true, then a warning message to rebuild the kernel is displayed and also recorded in the log file. However, *swremove* does not modify any of the kernel generation files.

- **Removing from a Distribution**

  The list of operations is simpler for removing filesets from a distribution than for installed_software.

  1. Remove each product:

     a. Remove each fileset in the product:

        i. Update the state of the fileset to `transient` in the catalog for the distribution.

        ii. Remove the files for the fileset.

        iii. Update the state of the fileset to `removed` in the catalog for the distribution or remove the catalog information for the fileset.

     b. If the catalog information has been removed for all filesets in the product, an implementation can also remove the catalog information for the product and its contained subproducts. For each fileset that failed to be removed, the distribution catalog information is updated to the state `corrupt` .

  2. Remove each bundle:

     a. Remove the distribution catalog information for the bundle.

  3. If the catalog information has been removed for all products and bundles in the distribution object, an implementation can also remove the catalog information for the distribution object.

- **Patches**

  Removal of the *base* fileset of a patch fileset via *swremove* will also result in removal of all patches to that fileset. Likewise when a particular base fileset is overwritten during a *swinstall* update operation, all patches for that base are removed as well. Saved rollback data is also removed when the base fileset to which it applies is updated or removed from the system.

  Removal of a patch automatically rolls back previous files, unless the rollback has been disabled as described for *swinstall* on page 90 or *swmodify* on page 108, or unless the base fileset is also being removed or updated. The patch fileset has the value of the *saved_files_directory* stored as an attribute, so that it can correctly find those files even if multiple *saved_files_directory* values have been used for various filesets.

  Only patches with the state `applied` can be rolled back. If a patch with a state of `committed` or `superseded` is attempted to be removed, generate an event. If the files for a patch do not exist in the *saved_files_directory*, or that directory no longer exists, generate an event.
  [SW_ERROR:SW_DEPENDENCY_NOT_MET]

  An installed patch that has been superseded may not be rolled back unless the superseding patch is also rolled back.

EXIT STATUS

See Chapter 3.

CONSEQUENCES OF ERRORS

See Chapter 3.

NAME

swverify — verify software

SYNOPSIS

`swverify [-d][-r][-F][-f file][-t` *`targetfile`*`][-x` *`option=value`*`]`
`[-X` *`options_file`*`][`*`software_selections`*`][@` *`targets`*`]`

DESCRIPTION

The *swverify* utility checks the accuracy of software in distributions and installed_software. The utility checks the integrity of directory structures and the files. Discrepancies are reported on stderr along with a detailed explanation of the problem.

OPTIONS

The *swverify* utility supports the following options. Where there is no description, the description in Chapter 3.

`-d`

-f file

`-F`

Correct problems as well as report them.

If *check_permissions=true*, correct the corresponding problems reported.

If *check_scripts=true*, correct the corresponding problems reported.

The `-F` option only applies to installed software.

`-r`

`-t` *targetfile*

`-x` *option=value*

`-X` *options_file*

OPERANDS

The *swverify* utility supports the *software_selections* and *targets* operands described in Chapter 3.

This utility need not support a target distribution in the serial format.

EXTERNAL INFLUENCES

See Chapter 3. for descriptions of external influences common to all utilities.

Extended Options

The *swverify* utility supports the following extended options. The description in Chapter 3 applies.

allow_incompatible=false

autoselect_dependencies=true

check_contents=true

check_permissions=true

check_requisites=true

check_scripts=true

check_volatile=false

distribution_target_directory=implementation_defined_value

enforce_dependencies=true

enforce_locatable=true

installed_software_catalog=implementation_defined_value

logfile=implementation_defined_value

loglevel=1

select_local=true

software=

targets=

verbose=1

EXTERNAL EFFECTS

See Chapter 3.

EXTENDED DESCRIPTION

See Chapter 3 for general information. The key phases in the *swverify* utility are:

1. Selection phase
2. Analysis phase
3. Execution phase.

Selection Phase

Like *swremove*, software selections apply to the software installed (or available in the case of a distribution).

Each specified selection is added to the selection list after it passes the following checks:

- If the selection is not found, generate an event.
  [SW_WARNING: SW_SELECTION_NOT_FOUND]
- If the selection is not found at that product location, but that product exists at another location, generate an event.
  [SW_WARNING: SW_SELECTION_NOT_FOUND_RELATED]

Add any dependencies to the selection list if *autoselect_dependencies=true*.

Analysis Phase

This section details the analysis phase for *swverify*. No aspect of the target host environment is modified unless the `-F` option is specified. The target role accesses its software_collection catalog to get the information for the selected software.

The target role makes the following checks:

- An event is generated for each product that is incompatible with the uname attributes of the target host. See Section 3.4.1.2 on page 42. If *allow_incompatible=false*, generate an event.
  [SW_ERROR: SW_NOT_COMPATIBLE]

  If *allow_incompatible=true*, generate an event.
  [SW_WARNING: SW_NOT_COMPATIBLE]

Applies to installed software.

- An event is generated for each fileset whose state is other than *installed*, *configured*, *available*, or *removed*.
  [SW_WARNING: SW_SELECTION_IS_CORRUPT]

  Applies to distributions and installed software.

- An event is generated if a dependency can not be met. If *enforce_dependencies=true*, generate an event.
  [SW_ERROR: SW_DEPENDENCY_NOT_MET]

  If *enforce_dependencies=false*, generate an event.
  [SW_WARNING: SW_DEPENDENCY_NOT_MET]

  Applies to distributions and installed software.

- Executes vendor-supplied verify scripts, generating an event if a verify script returns either an error or a warning.
  [SW_ERROR: SW_CHECK_SCRIPT_ERROR]
  [SW_WARNING: SW_CHECK_SCRIPT_WARNING]

  Applies to installed software.

- The following file level checks are made:

  — Check for missing files and directories. For installed software, if *check_volatile=false*, then this check must not be made for files with *file.is_volatile=true*.

    Applies to distributions and installed software.

  — Check for files that have been modified.

    For distributions,
    check *size*, *cksum*, and *mtime*.

    For installed software,
    check *mode*, *owner*, *group*, *size*, *cksum*, *mtime*, *revision*, *major*, and *minor*, if defined for that file object.
    If *check_volatile=false*, then these checks must not be made for files with *file.is_volatile=true*.

  — If a file is compressed, then the *compressed_size* and *compressed_cksum* attributes of the file should be checked instead of the *size* and *cksum* attributes.

    Applies to distributions.

  — Check symbolic links for correct values.

    Applies to distributions and installed software.

  If any of these checks fail for any file, generate an event.
  [SW_ERROR: SW_FILE_ERROR]

For patches, the verify operation on a patched fileset will check that the patched files are properly installed. When installing a patch, the ancestor fileset will be updated to have the correct attributes of the patched files. Verification of a patch fileset will verify that files in a patch are still properly installed (or in the depot correctly). For installed patches that have the fileset *saved_files_directory* defined, the saved files in the *saved_files_directory* will also be verified. This verification ensures that a patch can still be successfully rolled back.

Execution Phase

If the `-F` option is set, then the execution phase operations are run.

- **Execute Fix Scripts**

  In this step, *swverify* executes vendor-supplied fix scripts if operating on installed software.

  Scripts are executed in the same order as verify scripts. If a fix script returns an error, generate an event.
  [SW_ERROR: SW_PRE_SCRIPT_ERROR]

  If a fix script returns a warning, generate an event.
  [SW_WARNING: SW_PRE_SCRIPT_WARNING]

  Control scripts adhere to the specifications in section Section 3.6.1 on page 56.

- **File Level Fix**

  The following file level fixes are made:

  — Missing directories are created (except volatile unless the *check_volatile* option is true)

  — Files that have been modified (except volatile unless the *check_volatile* option is true) are fixed for *mode*, *owner*, *group*, *major*, and *minor*, as applicable

  — Symbolic links are re-created to correct their values.

  — Any files with type x (delete file) as a new item are removed.

  If any of these fixes fail for any file, generate an event.
  [SW_ERROR: SW_FILE_ERROR]

EXIT STATUS

See Chapter 3.

CONSEQUENCES OF ERRORS

See Chapter 3.

Chapter 5

Software Packaging Layout

This section describes the software packaging layout. The software packaging layout consists of:

1. The directory structure consisting of these major components:
   - The exported catalog structure containing software information including software definition files and customize scripts used by the install/update and copy utilities
   - The file storage structure that contains the actual software files for each fileset
2. the software definition file formats and the objects and attributes they contain, `INDEX` for software definitions and `INFO` for file and control_file definitions (used by all the utilities defined in this Software Administration specification), the PSF for product specification (used by *swpackage*), and the `space` file (used by disk space analysis in install).
3. The serial format of the layout containing an archive of files in the directory structure.

   This ordering of directories and files also applies to distributions on a set of hierarchical file systems which span multiple media.

Thus, two distinct (but related) formats for the software packaging layout are supported by this Software Administration specification:

- A directory structure format which resides within a POSIX.1 hierarchical file system (disk, CD-ROM, etc.)
- A bit stream serial format which resides within a POSIX.1 extended *cpio* or extended *tar* archive

A conformant implementation does not contain any other files or directories besides those explicitly entered in the distribution catalog. However, it can contain other files and directories besides those belonging to the distribution.

5.1 Directory Structure

This section describes the directory structure for the software packaging layout, and how this representation stores the definitions of the software and file objects contained within it. The directory structure is a POSIX.1 hierarchical file system containing files in the software packaging layout.

This Software Administration specification defines a single directory structure for directory and serial distributions. The software packaging layout can be stored in two forms:

- A direct access file system as described in this section
- A serial access extended *cpio* or extended *tar* archive as described in POSIX.1

The same structure offers optimal load performance for serial distributions while providing a simple structure for directory distributions. This structure applies to each media if the distribution spans multiple media.

The structure supports multiple versions of a product contained within a single distribution, where versions are distinguished by a unique combination the product *tag*, *revision*, *architecture*, and *vendor_tag* attributes.

Table 5-1 shows the directory structure of a software packaging layout located under a directory *path*:

The directory structure for the software packaging layout is divided into two areas:

- The exported catalog structure, consisting of control directories containing the software definition files that describe the products contained in the distribution, as well as the software control_files
- The software file storage structure, consisting of the product and fileset storage directories, under which the actual software files for each fileset are located

5.1.1 Exported Catalog Structure

The catalog structure describes the software contained in the distribution. It is organized by product, and each product is organized by fileset. The specific contents are described in the following sections.

5.1.1.1 INDEX File

The distribution catalog contains a global `INDEX` file:

- `catalog/INDEX`

  This `INDEX` file contains the definition of all software objects in the distribution.

5.1.1.2 Distribution Files

The `catalog/dfiles/` directory contains files used to store certain attributes of the distribution object. The distribution information stored can include:

- *<attribute>*

  A distribution attribute can be stored as a separate file, the file name of which can be the name of the attribute.

Table 5-1 Example of Software Packaging Layout

Directory or File	**Purpose**
path/catalog/ *path*/catalog/INDEX *path*/catalog/dfiles/ *path*/catalog/dfiles/...	Contains all information about the distributions contents Global index of distribution and its contents Contains distribution attributes stored in files
path/catalog/*product1*/ *path*/catalog/*product1*/pfiles *path*/catalog/*product1*/pfiles/INFO *path*/catalog/*product1*/pfiles/*script* *path*/catalog/*product1*/pfiles/...	Storage for information on the first product Contains all product attributes stored in files Control_file information for this product Vendor defined control_files
path/catalog/*product1*/*fileset1* *path*/catalog/*product1*/*fileset1*/INFO *path*/catalog/*product1*/*fileset1*/*scripts* *path*/catalog/*product1*/*fileset1*/...	Storage for information and scripts on this fileset File and control_file information for this fileset Vendor defined control_files
path/catalog/*product1*/*fileset2*/ *path*/catalog/*product1*/*fileset2*/...	Storage for information and scripts on the next fileset
path/catalog/*product2*/ *path*/catalog/*product2*/...	Storage for information on the next product
path/*product1*/ *path*/*product1*/*fileset1*/ *path*/*product1*/*fileset1*/*files* *path*/*product1*/*fileset1*/...	Storage for this product's filesets Storage for this filesets files Actual directory structure of files
path/*product1*/*fileset2*/ *path*/*product1*/*fileset2*/*files* *path*/*product1*/*fileset2*/...	Storage for next fileset's files Actual directory structure of files
path/*product2*/ *path*/*product2*/...	Storage for next product's filesets

5.1.1.3 Product Catalog

The catalog files for each product are stored under a directory `catalog/`<***product_control_directory***>`/`. The way in which the value of each product control directory is determined is defined below.

5.1.1.4 Product Control Files

The `catalog/`<***product_control_directory***>`/pfiles/` directory contains the control_files for the product object. The product control_files include:

- <***attribute***>

  A product attribute can be stored as a separate file, the file name of which can be the name of the attribute.

- `INFO`

  Contains the definitions for the control_file objects contained within the product.

- `checkinstall`
  `preinstall`
  ...
  `postremove`

  The vendor-supplied control scripts for the product.

- *<control_file>*

  All other vendor-defined control_files for this product.

5.1.1.5 *Fileset Control Files*

The `catalog/`*<product_control_directory>*`/`*<fileset_control_directory>* directory contains the control_files for the fileset object. The way in which the value of each fileset control directory is determined is defined below. The fileset control_files include:

- *<attribute>*

  A fileset attribute can be stored as a separate file, the filename of which can be the name of the attribute.

- `INFO`
  Contains the definitions for the control_file and file objects contained within the fileset.

- `checkinstall`
  `preinstall`
  ...
  `postremove`

  The vendor-supplied control scripts for the fileset.

- *<control_file>*

  All other vendor-defined control_files for this fileset.

5.1.2 File Storage Structure

The second portion of a distribution contains the actual software files contained in each fileset object.

The files of each fileset are store in a directory with the name *<fileset_control_directory>* which is itself in a directory called *<product_control_directory>*.

Each regular file (ones for which *file.type* is `f`) is stored in a location defined by appending the *file.path* attribute to the path of the fileset file storage directory. This may require the creation of additional directories. Other file types (directories, except as needed to store files, hard links and symbolic links) are not required to exist in the distribution. The POSIX.1 file permissions for files in the file storage area are undefined.

5.1.2.1 Control Directory Names

In the simplest case, the value of the *product.tag* attribute is the name of the product control directory. The *fileset.tag* attribute is used as the name of the fileset control directory. Two conditions complicate this simple naming:

1. Length of the tag attribute exceeds {POSIX_PATH_MAX} of the system where the distribution resides.
2. Name collision with an existing product control directory.

   Given that multiple versions of a product may be contained in the same distribution, collisions from product control directories named by the tag attribute are common.

These conditions are met by defining a *control_directory* attribute for each product and fileset that is unique within the distribution. The attribute uses the syntax:

```
%token          FILENAME_CHARACTER_STRING /* as defined in 2.2.2.37  */

%start  control_directory
%%

control_directory       : tag_part
                        | tag_part "."  instance_id_part
                        ;

tag_part                : FILENAME_CHARACTER_STRING
                        ;

instance_id_part        : FILENAME_CHARACTER_STRING
                        ;
```

The `tag_part` may be the product or fileset *tag* attribute, truncated as necessary to meet any filename length restrictions of the operating system.

The `instance_id_part` is a string that, when added after the ‘‘.’’ (period), defines a `control_directory` which, for products, is unique within the distribution and, for filesets, is unique within the product. For products, this `instance_id_part` may be the *instance_id* of the product if that *instance_id* was generated considering other products `tag_parts` in addition to *tag* attributes.

5.2 Software Definition File Format

The software definition files contain the software structure and the detailed attributes for distributions, installed_software, bundles, products, subproducts, filesets, files, and control_files. While information on installed software is represented in this form as input to or output from the various software administration utilities, the actual storage of this metadata for installed software is undefined. This section describes the format of the software definition files:

- The `INDEX` file contains the definition of distribution or installed_software objects as well as the software objects contained within those software_collections. The information in this file is primarily used in selection phases of the utilities.
- The `INFO` file contains the definition of the software files and control_files for a product or fileset within a distribution or installed_software object. The information in this file is primarily used in analysis and execution phases of the utilities.
- The PSF (product specification file) also contains the definition of distribution attributes, software objects, and the software files and control_files for the product and fileset software objects. This file is created by the software vendor and used by the packaging tool to create the distribution, represented by the `INDEX` and `INFO` files, in the software packaging layout.

  The PSF supports the same syntax as the `INDEX` and `INFO` files. Additional syntactic constructs are supported for specifying files and control_files. This file is used in selection, analysis and packaging Phases of the *swpackage* command.

Additionally, there is a `space` file that is created by the software vendor for additional disk space needed for a product or fileset. This file is used in analysis phase of the *swinstall* command to account for additional disk space required.

5.2.1 Software Definition File Syntax

The `INDEX` and `INFO` files have essentially the same syntax and semantics as the PSF. One key difference is that the `INDEX` file does not contain control_file and file definitions, the `INFO` file contains only control_file and file definitions, and the PSF file contains all definitions. In a distribution, each product and fileset has a separate `INFO` file.

The software specification file syntax is as follows.[16]

```
%token          FILENAME_CHARACTER_STRING  /* as defined in Glossary */
%token          NEWLINE_STRING             /* as defined in Glossary */
%token          PATHNAME_CHARACTER_STRING  /* as defined in Glossary */
%token          SHELL_TOKEN_STRING         /* as defined in Glossary */
%token          WHITE_SPACE_STRING         /* as defined in Glossary */


%start  software_definition_file
%%

software_definition_file  : INDEX
                          | INFO
                          | PSF
                          ;
```

16. Refer to Appendix A for examples of the use of this syntax.

```
INDEX                     : soc_definition
                            soc_contents
                          ;

INFO                      : info_contents
                          ;

PSF                       : distribution_definition
                            soc_contents
                          ;


media                     : /* empty */
                          | media_definition
                          ;

vendors                   : /* empty */
                          | vendors NEWLINE_STRING vendor_definition
                          | vendor_definition
                          ;


bundles                   : /* empty */
                          | bundles NEWLINE_STRING bundle_definition
                          | bundle_definition
                          ;

products                  : /* empty */
                          | products NEWLINE_STRING product_specification
                          | product_specification
                          ;

product_specification     : product_definition
                            product_contents
                          ;

subproducts               : /* empty */
                          | subproducts NEWLINE_STRING subproduct_definition
                          | subproduct_definition
                          ;

filesets                  : filesets NEWLINE_STRING fileset_specification
                          | fileset_specification
                          ;

fileset_specification     : fileset_definition
                            fileset_contents
                            /* fileset contents not valid in INDEX files */
                          ;

control_files             : /* empty */
                          | control_files NEWLINE_STRING control_file_definition
                          | control_file_definition
                          ;

files                     : /* empty */
                          | files NEWLINE_STRING file_definition
                          | file_definition
                          ;
```

```
fileset_contents            : fileset_contents NEWLINE_STRING fileset_content_items
                            | fileset_content_items
                            ;

fileset_content_items       : control_files
                            | files
                            ;

info_contents               : info_contents NEWLINE_STRING info_content_items
                            | info_content_items
                            ;

info_content_items          : control_files
                            | files
                            ;

product_contents            : product_contents NEWLINE_STRING product_content_items
                            | product_content_items
                            ;

product_content_items       : control_files
                              /* control_files not valid in INDEX files */
                            | subproducts
                            | filesets
                            ;

soc_contents                : soc_contents NEWLINE_STRING soc_content_items
                            | soc_content_items
                            ;

soc_content_items           : vendors
                            | bundles
                            | products
                            ;

soc_definition              : distribution_definition
                            | installed_software_definition
                            ;

distribution_definition     : software_definition
                              media
                            ;

media_definition            : software_definition
                            ;

installed_software_definition  : software_definition
                               ;

vendor_definition           : software_definition
                            ;

bundle_definition           : software_definition
                            ;

product_definition          : software_definition
                            ;

subproduct_definition       : software_definition
```

```
                          ;

fileset_definition        : software_definition
                          ;

control_file_definition   : software_definition
                          | extended_definition
                            /* extended_definition only valid in PSF files */
                          ;

file_definition           : software_definition
                          | extended_definition
                            /* extended_definition only valid in PSF files */
                          ;

software_definition       : object_keyword NEWLINE_STRING
                            attribute_value_list
                          ;

attribute_value_list      : /* empty */
                          | attribute_value_list attribute_definition NEWLINE_STRING
                          | attribute_definition NEWLINE_STRING
                          ;

attribute_definition      : attribute_keyword WHITE_SPACE_STRING attribute_value
                          ;

object_keyword            : FILENAME_CHARACTER_STRING
                          ;

attribute_keyword         : FILENAME_CHARACTER_STRING
                          ;

extended_definition       : extended_keyword WHITE_SPACE_STRING attribute_value
                          ;

extended_keyword          : FILENAME_CHARACTER_STRING
                          ;

attribute_value           : attribute_value WHITE_SPACE_STRING single_value
                          | single_value
                          | '<' WHITE_SPACE_STRING PATHNAME_CHARACTER_STRING
                          | '<' PATHNAME_CHARACTER_STRING
                          ;

single_value              : SHELL_TOKEN_STRING
                          ;
```

The following syntax rules are applicable to software definition files:

1. All keywords and values are represented as character strings.
2. Each keyword is located on a separate line. Keywords can be preceded by white space (tab, space). White space separates the keyword from the value.
3. Comments can be placed on a line by themselves or after the keyword-value syntax. They are designated by preceding them with the `#` (pound) character. The way in which comments are used in `INDEX` and `INFO` is undefined

4. All object keywords have no values. All attribute keywords have one or more values.
5. An attribute value ends on the same line as the keyword with one exception. Attribute values can span lines if and only if the value is prefixed and suffixed with the `"` (double quote) character.
6. When an attribute value begins with `<` (less than), the remainder of the string value is interpreted as a filename whose contents will be used as a quoted string value for the attribute. For `INDEX` files, the filename is a path relative to the control directory for that distribution, product, or fileset. For PSF files, the filename is a path to a file on the host that contains the file.
7. The use of `"` (double quote) is not required when defining a single line string value that contains embedded white space. Trailing white space is removed; embedded white space is used. The quotes can be used.
8. The `"` (double quote), `#` (pound), and `\` (backslash) characters can be included in multi-line string values by ''escaping'' them with `\` (backslash).
9. The order of attributes is not significant, except that the *layout_version* is the first attribute defined in an `INDEX` file for a distribution or installed_software object.

5.2.1.1 Keyword and Attribute Semantics

The keywords and attribute types have the following semantics:

1. The object keywords **distribution**, **installed_software**, **media**, **category**, **vendor**, **bundle**, **product**, **subproduct**, **fileset**, **control_file**, and **file** each define a new object of that type. The keywords **distribution**, **installed_software**, **product**, and **fileset** also define nested blocks that contain the objects describing the software hierarchy.
2. If an attribute is not supplied, then its default value is used, unless no default value is permitted.
3. Attributes that have boolean permitted values are described by the strings `true` and `false`.
4. Attributes that have an enumerated set of permitted values are described by one of the enumerated values. Enumerated values do not contain spaces and are case sensitive. In addition, abbreviations of the string are not allowed. For example, `conf` is not equivalent to `configured`.
5. For attributes whose values are integer character strings, the default value is used if the attribute is not supplied. If the first two characters of an integer character string are `0x` (zero followed by a lowercase ''x''), then the value is interpreted as hexadecimal. Otherwise, if the first character of an integer character string is `0` (zero), then the value is interpreted as octal. Attribute values denoting time are integer character strings that signify seconds since the Epoch.
6. Attributes whose permitted values are lists of *tag*s or `software_spec`s can be described either by one or more repeating keywords, each listing one or more *tag*s or `software_spec` s separated by white space (for example, for *subproduct.contents* or *fileset.prerequisites*), or by blocks of object fragments (for example, product, fileset, and file definitions).

   The former is used when the hierarchy is defined by reference, and the latter when the hierarchy is defined by containment. For example, subproducts and filesets are contained within products, but filesets are referenced by subproducts.

7. Attributes that have permitted values of `software_pattern_matching_string` are software pattern matching strings as described in the Glossary. For all product attributes related to the *uname* structure (as defined in POSIX.1, an empty string value is treated as equivalent to `*` (asterisk), implying a universal match.

5.2.1.2 Vendor Defined Keywords and Attributes

A software definition file can contain keywords (implying attributes) not defined by this Software Administration specification. All such keywords in a file not recognized by an implementation are preserved (along with their associated values) by being transferred to the resulting `INDEX` or `INFO` files created by *swpackage* or *swcopy*. For any keyword, the keyword itself is a filename character string.

The value associated with any keyword is processed as an `attribute_value` (see Section 5.2.1 on page 130) and thus can be continued across multiple input lines or can reference a file containing the value for the keyword.

Implementations that make use of keywords beyond those described in this Software Administration specification take actions they believe appropriate for those keywords. The handling of any keywords which are both not defined by this Software Administration specification, and still recognized by an implementation, is undefined.

5.2.2 Distribution Definition

```
distribution
    layout_version          layout_version
    path                    path
    dfiles                  dfiles
    pfiles                  pfiles
    uuid                    uuid
```

`INDEX` and PSF files can contain distribution definitions. Neither file contains the *path* attribute. Its value is generated dynamically by *swlist*.

The *bundles, media*, and *products* attributes are not stored as attributes, but rather as **bundle**, **media**, and **product** definitions. These attributes are not included in *swlist* `-v` output. Rather, they are generated dynamically only by *swlist* `-a` *attribute*.

A PSF does not require a distribution definition. The PSF does not contain the *uuid* attribute. It is generated dynamically, if needed, by *swcopy* and *swpackage*.

An `INDEX` file contains the *layout_version* attribute as the first attribute defined in the file. Distributions which span multiple media contains the *uuid* attribute.

5.2.3 Media Definition

```
media
    sequence_number    sequence_number
```

`INDEX` files for distributions can contain media definitions.

An `INDEX` file for a distribution contains the *sequence_number* attribute if the distribution spans multiple media.

5.2.4 Installed Software Definition

```
installed_software
    layout_version    layout_version
    path              path
    dfiles            dfiles
    pfiles            pfiles
    catalog           catalog
```

The storage of catalogs for installed software is undefined. With the use of the *swlist* utility, the contents of such catalogs may be manifested in exported catalog form. The rules contained within this section applies when the contents of an installed software catalog is manifested in exported catalog form.

`INDEX` files can contain installed_software definitions. This describes the attributes for installed_software objects when listed in exported catalog structure using *swlist*.

The *products* and *bundles* attributes are not stored as attributes, but rather as product and bundle definitions. These attributes are not included in *swlist* `-v` output. Rather, they are generated dynamically only by *swlist* `-a` *attribute* .

An `INDEX` file does not contain the *path* or *catalog* attributes; they are generated dynamically by *swlist*.

An `INDEX` file contains the *layout_version* attribute as the first attribute defined in the file.

5.2.5 Vendor Definition

```
vendor
    tag            tag
    title          title
    description    description
```

`INDEX` and PSF files can contain vendor definitions. The *tag* attribute is required for all vendor objects.

5.2.6 Category Definition

```
category
    tag            tag
    title          title
    description    description
    revision       revision
```

`INDEX` and `PSF` files can contain category definitions. The *tag* attribute is required for all category objects.

5.2.7 Bundle Definition

bundle	
tag	*tag*
architecture	*architecture*
category_tag	*category_tag*
location	*location*
qualifier	*qualifier*
revision	*revision*
vendor_tag	*vendor_tag*
contents	*contents*
copyright	*copyright*
create_time	*create_time*
description	*description*
directory	*directory*
instance_id	*instance_id*
is_locatable	*is_locatable*
is_locatable	*is_locatable*
is_patch	*is_patch*
machine_type	*machine_type*
mod_time	*mod_time*
number	*number*
os_name	*os_name*
os_release	*os_release*
os_version	*os_version*
size	*size*
title	*title*

`INDEX` and PSF files can contain bundle definitions. The *tag* and *contents* attributes are required for all bundles.

Neither file contains the *size* attribute. The value of the *size* attribute is generated dynamically based on the sizes of the filesets currently contained within the bundle.

An `INDEX` file also contains an *instance_id* attribute. The value of the *instance_id* attribute is generated dynamically by *swpackage* or *swcopy*. An `INDEX` file for installed software contains a *create_time* attribute and a *mod_time* attribute for each bundle.

Only bundle definitions for installed software may contain either the *location* or *qualifier* attributes; bundle definitions for distributions do not contain either the *location* or *qualifier* attributes.

A PSF should not contain either the *location* or *qualifier* attributes; they are ignored when parsing the file.

5.2.8 Product Definition

```
product
    tag                   tag
    architecture          architecture
    category_tag          category_tag
    qualifier             qualifier
    revision              revision
    vendor_tag            vendor_tag
    all_filesets          all_filesets
    control_directory     control_directory
    copyright             copyright
    create_time           create_time
    directory             directory
    description           description
    instance_id           instance_id
    is_locatable          is_locatable
    is_patch              is_patch
    postkernel            postkernel
    location              location
    machine_type          machine_type
    mod_time              mod_time
    number                number
    os_name               os_name
    os_release            os_release
    os_version            os_version
    size                  size
    title                 title
```

`INDEX` and PSF files can contain product definitions. The *tag* and *control_directory* attributes are required for all products.

Neither file contains the *size* attribute. The value of the *size* attribute is generated dynamically based on the sizes of the filesets currently contained within the product.

An `INDEX` file for installed software contains a *create_time* attribute and a *mod_time* attribute for each product.

The *control_files*, *filesets*, and *subproducts* attributes are not stored as attributes, but rather as **control_file**, **fileset**, and **subproduct** definitions. These attributes are not included in *swlist* `-v` output. Rather, they are generated only by *swlist* `-a` *attribute* .

An `INDEX` file contains an *instance_id* attribute. The value of the *instance_id* attribute is generated by *swpackage* or *swcopy*. An `INDEX` file also contains a *all_filesets* attribute in addition to the fileset definitions. This attribute is generated by *swpackage* and represents all filesets defined for the product, as opposed to those that are currently contained within the product. The value of the *filesets* and *all_filesets* attributes may differ, since some originally defined filesets might not be copied or installed. Only product definitions for installed software may contain either the *location* or *qualifier* attributes; product definitions for distributions do not contain either the *location* or *qualifier* attributes.

A PSF should not contain either the *location* or *qualifier* attributes; they are ignored when parsing the file.

5.2.9 Subproduct Definition

subproduct	
tag	*tag*
contents	*subproducts*
category_tag	*category_tag*
create_time	*create_time*
description	*description*
is_patch	*is_patch*
mod_time	*mod_time*
size	*size*
title	*title*

INDEX and PSF files can contain subproduct definitions. The *tag* and *contents* attributes are required for all subproducts.

Neither file contains the *size* attribute. The value of the *size* attribute is generated dynamically based on the sizes of the filesets currently contained within the subproduct.

An INDEX file for installed software contains a *create_time* attribute and a *mod_time* attribute for each subproduct.

5.2.10 Fileset Definition

fileset	
tag	*tag*
ancestor	*ancestor*
applied_patches	*applied_patches*
category_tag	*category_tag*
control_directory	*control_directory*
corequisites	*corequisites*
create_time	*create_time*
description	*description*
exrequisites	*exrequisites*
is_reboot	*is_reboot*
is_kernel	*is_kernel*
is_locatable	*is_locatable*
is_patch	*is_patch*
is_sparse	*is_sparse*
location	*location*
media_sequence_number	*media_sequence_number*
mod_time	*mod_time*
patch_state	*patch_state*
prerequisites	*prerequisites*
revision	*revision*
saved_files_directory	*saved_files_directory*
size	*size*
state	*state*
superseded_by	*superseded_by*
supersedes	*supersedes*
title	*title*

INDEX and PSF files can contain fileset definitions. The *tag* and *control_directory* attributes are required for all filesets.

The *control_files* and *files* attributes are not stored as attributes, but rather as **control_file** and **file** definitions. These attributes are not included in *swlist* `-v` output. Rather, they are generated only by *swlist* `-a` ***attribute*** .

An `INDEX` file contains a *size* attribute for each defined fileset. Fileset definitions for distributions that span multiple media contain the *media_sequence_number* attribute. An `INDEX` file for installed_software contains a *create_time* and a *mod_time* attribute for each fileset.

A PSF should not contain the *applied_patches*, *location*, *media_sequence_number*, *patch_state*, *saved_files_directory*, *size*, *state*, or *superseded_by* attributes; they are ignored when parsing the file. The value of the *size* attribute is generated dynamically by *swpackage* based on the sizes of the files and control_files.

When defining a patch, the attributes that must be defined are the files to be patched, and the fileset attribute *is_patch*. This automatically sets the category_tag *patch*; the *patch* category can not be specified in a PSF file. The product containing a *patch* or *sparse* fileset must have a different tag or revision attribute from the product containing the ancestor fileset being updated.

5.2.11 Control_File Definition

```
control_file
    tag                 tag
    cksum               cksum
    compressed_cksum    compressed_cksum
    compressed_size     compressed_size
    compression_state   compression_state
    compression_type    compression_type
    interpreter         interpreter
    path                path
    revision            revision
    size                size
    source              source
    result              result
```

`INFO` and PSF files can contain control_file definitions.

A PSF should not contain the *cksum*, *compressed_cksum*, *compressed_size*, *compression_state*, *compression_type*, *size*, or *result* attributes; they are ignored when parsing the file. The values of the *size* and *cksum* attributes are generated dynamically by *swpackage* based on the source file. A PSF contains a *source* attribute. A PSF can contain a *path* attribute. If it does not, *swpackage* uses the basename obtained from the value of the *source* attribute as the value of the *path* attribute. A PSF can contain a *tag* attribute. If it does not, *swpackage* uses the basename obtained from the value of the *path* attribute as the value of the *tag* attribute. The *swpackage* utility resolves the value of the *tag* attribute after it resolves the value of the *path* attribute.

An `INDEX` file contains the *tag*, *path*, *cksum*, and *size* attributes. Control_file definitions for installed software also contain the *result* attribute. `INDEX` files should not contain the *source* attribute; it is ignored when parsing the file.

The *swpackage* command automatically includes the `INFO` file itself as a control file and adds the *tag*, *path*, and *size* attributes for it. The value of the *cksum* attribute for the `INFO` control_file itself is not defined. An implementation can choose to store certain software object attributes, such as *copyright*, as control_files.

5.2.12 File Definition

```
file
    path                path
    archive_path        archive_path
    cksum               cksum
    compressed_cksum    compressed_cksum
    compressed_size     compressed_size
    compression_state   compression_state
    compression_type    compression_type
    gid                 gid
    group               group
    is_volatile         is_volatile
    link_source         link_source
    major               major
    minor               minor
    mode                mode
    mtime               mtime
    owner               owner
    revision            revision
    size                size
    source              source
    type                type
    uid                 uid
```

`INFO` and PSF can contain file definitions.

A PSF contains a *source* attribute. A PSF should not contain the *cksum* *compressed_cksum*, *compressed_size*, *compression_state*, *compression_type*, or *size* attributes; they are ignored when parsing the file. Device files (including the *major* and *minor* attributes) should not be defined in a PSF (but can be added via *swmodify* after being created by a `configure` script); they are ignored when parsing the file. The values of the *size* and *cksum* attributes are generated dynamically by *swpackage* based on the source file. A PSF can contain a *path* attribute, otherwise the source is used to defined the path by *swpackage*. A PSF can contain *gid*, *group*, *link_source*, *mode*, *mtime*, *owner*, *type*, and *uid* attributes, otherwise they are retrieved from the source file by *swpackage*. A PSF can contain *is_volatile* and *revision* attributes. Automatic determination of the file revision is implementation defined.

An `INFO` file should not contain the *source* attribute; it is ignored when parsing the file. Table 5-2 shows the required, optional and non-applicable attributes for each of the file types in an `INFO` file. The file types are described in Table 2-14 on page 33. Within a fileset, no more than one copy of a file is stored with the same path.

Within a PSF file, if the same file is defined more than once, the attributes from the last definition are used and they redefine the attributes previously defined. This action does not cause additional copies of the file to be stored in the distribution. All attributes not specifically listed remain unchanged.

Table 5-2 File Attributes for INFO File

attribute	f	d	h	s	b	c	p	x	a
type	R	R	R	R	R	R	R	R	R
path	R	R	R	R	R	R	R	R	R
size	R	-	-	-	-	-	-	O	R
link_source	-	-	R	R	-	-	-	-	-
mode	O	O	-	-	O	O	O	-	O
owner	O	O	-	-	O	O	O	-	O
group	O	O	-	-	O	O	O	-	O
uid	O	O	-	-	O	O	O	-	O
gid	O	O	-	-	O	O	O	-	O
cksum	O	-	-	-	-	-	-	-	O
major	-	-	-	-	R	R	-	-	-
minor	-	-	-	-	R	R	-	-	-
is_volatile	O	O	O	O	O	O	O	-	O
mtime	O	-	-	-	-	-	-	-	O
revision	O	-	-	-	-	-	-	-	O
archive_path	-	-	-	-	-	-	-	-	R

Key: R Required
O Optional
- Ignored

5.2.13 Extended Control_File Definitions

```
checkinstall     source [path]
preinstall       source [path]
postinstall      source [path]
verify           source [path]
fix              source [path]
checkremove      source [path]
preremove        source [path]
postremove       source [path]
configure        source [path]
unconfigure      source [path]
request          source [path]
unpreinstall     source [path]
unpostinstall    source [path]
space            source [path]
control_file     source [path]
```

A PSF can contain extended control_file definitions. Each control_file definition defines the *source* attribute (the source file) to be used for the script. The keyword (meaning **checkinstall**, **preinstall**, etc.) defines the *tag* of the script, which tells the utilities when to execute the script.

If the optional *path* is supplied, it is the file name in the distribution (relative to the control directory for the software containing this script) used to store the file; otherwise, the *control_file.tag* attribute is used as the file name. This also allows a vendor to define one script to be executed for multiple tags. The script can determine the *tag* being executed by the **SW_CONTROL_TAG** environment variable.

If the **control_file** keyword is used, then the basename of the *source* attribute is the *tag* of the control_file.

5.2.14 Extended File Definitions

A PSF can contain extended file definitions. The *swpackage* utility supports these extended file definition mechanisms:

directory mapping
: A PSF can point the *swpackage* utility at a source directory containing the files for the fileset. In addition, a PSF can map this source directory to the appropriate (destination) directory containing this subset of the filesets files.

recursive (implicit) file definition
: If a directory mapping is active, a PSF can direct the *swpackage* utility to include all files (recursively) from within the directory in the fileset.

explicit file definition
: For all or some of the files in the fileset, a PSF can name each source file and destination path with a one line per file syntax.

default permission definition
: For all or some of the files in the fileset, a PSF can define a default set of permissions.

excluding files
: Files that otherwise would be included can be explicitly excluded.

including files
: File definitions may be included from a separate file.

These mechanisms can all be used in combination with the others.

5.2.14.1 Directory Mapping

`directory` ***source*** `[`***path***`]`

This syntax defines a *source* directory under which subsequently listed files are located. In addition, the user can map the *source* directory to a *destination* directory under which the packaged files will be located.

The destination directory is an absolute pathname and is used as a prefix to the *path* attribute for each of the files.

The source directory can be either an absolute pathname, or a relative pathname. If relative, the *swpackage* utility interprets it relative to the current working directory in which the utility was invoked.

If the *source* directory does not exist, the *swpackage* utility generates an error.

5.2.14.2 Recursive File Definition

```
file  *
```

This syntax directs the *swpackage* utility to include every file (and directory) within the current source directory in the fileset. The *swpackage* utility attempts to include the entire, recursive contents of the source directory in the fileset.

The **directory** keyword is specified before the **file** * specification is used. After finishing the recursive processing of the source directory, the *swpackage* utility processes further specifications with respect to the original directory.

All other attributes for the destination file object are taken from the source file, unless a **file_permission** keyword is active. This keyword is described below.

The user can specify multiple **directory** and **file** * pairs to gather all files from different source directories into a single fileset.

5.2.14.3 Explicit File Definition

```
file [-t type][-m mode][-o owner[,uid]][-g group[,gid]][-n][-v]
     source [path]
```

Instead of, or in addition to, the recursive file specification, the user can explicitly specify the files to be packaged into a fileset.

This syntax may be used to redefine an attribute of a previously defined file. All attributes not specifically listed remain the same.

The **directory** keyword can be used to define a source (and destination) for explicitly specified files. If no **directory** keyword is active, then the full source path and the absolute destination path (the *path* attribute) is specified for each file.

The meaning of each of these fields is as follows:

file

This keyword specifies an existing file or directory, perhaps within the currently active source directory, to include in the fileset. It can also specify a directory, hard link, or symbolic link that does not exist as a source file, but is created when the fileset is installed.

source

When specifying an existing source file, this value defines the path to it.

If this is a relative path, the *swpackage* utility searches for it relative to the source directory set by the **directory** keyword. If no source directory is active, the *swpackage* utility searches for it relative to the current working directory in which the utility was invoked.

All attributes for the destination file object are taken from the source file, unless a **file_permission** keyword is active, or the `-m`, `-o`, or `-g` options are also included in the file specification.

When specifying a new directory to be created upon installation, and there is no destination *path* specified, the *source* defines the path of the installed directory. When specifying a new hard link or symbolic link to be created upon installation, the *source* defines the pathname of the installed file to use as the source for the new file.

path

When specifying a new or existing source file, this value defines the destination path at which the file will be created or installed. If *path* is a relative path, the active destination directory set by the **directory** keyword is prefixed to it. If the path is relative, and no destination directory is active, the *swpackage* utility generates an error. If the path is not specified, then the *source* is used as the *path*, with the appropriate mapping done with the active destination directory (if any).

`-t` *type*

When creating a new directory, hard link or symbolic link (a file in the fileset that does not exist in the source), this option is specified to define the file type. The following file types can be created:

`d` Create a directory. If only the *source* is specified, it is used as the *path*. Otherwise, the *source* is used to retrieve the attributes for the directory created at *path*. If the path is

not specified, or any attributes, then default values of the attributes is implementation defined.

h Create a hard link. Both the *source* and *path* are specified. The *source* is the pathname of the installed file object to be used as the source for the new hard link (the *link_source* attribute).

s Create a symbolic link. Both the *source* and *path* are specified. The *source* is the pathname of the installed file object to be used as the source for the new symbolic link (the *link_source* attribute).

x Delete a file. The *remove* type, *x* for the file *type* attribute deletes any existing file instead of installing a new file or link.

Files with the types c (character special), b (block special) and p (named pipe | FIFO) are not supported by *swpackage* and *swinstall* and can be created via a configure control script. In general, device files and pipes are created during system configuration on the system actually running the software. Also, there can be files of other types that the *swpackage* utility does not recognize and which therefore cause an error.

-m *mode*
This option defines the (octal) mode for a file or directory.

-o [*owner*][,*uid*]
This option defines the owner name and/or or uid of the destination file. If only the *owner* is specified, then the *owner* and *uid* attributes are set for the destination file object, based on the database of the packaging host. If only the *uid* is specified, it is set as the *uid* attribute for the destination object and no owner name is assigned. If both are specified, each sets the corresponding attribute for the file object. If neither are specified, then the owner and uid of the file are used as found in the file system of the packaging host. See ‘‘File Loading’’ in the definition for *swinstall swinstall* on page 90.

During an installation, the *owner* attribute is used to set the owner name and uid, unless the owner name is not defined in the target system user database. In this case, the value of the *uid* attribute is used to set the uid.

-g [*group*][,*gid*]
This option defines the group name and/or gid of the destination file. If only the *group* is specified, then the *group* and *gid* attributes are set for the destination file object based on the database of the packaging host. If only the *group* is specified, and it contains digits only, it is interpreted as the gid, and is set as the *gid* attribute for the destination object; no group name is assigned to the object. If both are specified, each sets the corresponding attribute for the file object. If neither are specified, then the group and gid of the file are used as found in the file system of the packaging host. See ‘‘File Loading’’ in the definition for *swinstall swinstall* on page 90.

During an installation, the *group* attribute is used to set the group name and gid, unless the group name is not defined in the target system group database. In this case, the *gid* attribute is used to set the gid.

-n This option indicates that the file is not compressible.

-v The use of -v on a source line is used to specify that the file is volatile (contents, attributes or existence can change after installation).

5.2.14.4 *Default Permission Definition*

```
file_permissions [-m mode | -u umask][-o [owner[,]][uid]]
                 [-g [group[,]][gid]]
```

A destination file object inherits the mode, owner, and group of the source file. The **file_permissions** keyword can be specified to set a default permission mask, owner, and group for all the files being packaged into the fileset:

`file_permissions`
: This keyword only applies to the fileset it is defined in. Multiple **file_permissions** can be specified, and subsequent definitions simply replace previous definitions.

`-m` *mode*
: This option defines a default (octal) mode for all file objects.

`-u` *umask*
: Instead of specifying an octal mode as the default, the user can specify an octal *umask*() value that gets ''subtracted'' from the mode of an existing source file, or applied for each non-existent file, to generate the mode of the destination file.

  By specifying a *umask*() value the user can set a default mode for executable files, non-executable files, and directories. A specific mode can be set for any file, as described above.

`-o [`*owner*`[,]][`*uid*`]`
: This option defines the owner name and/or uid of the destination file. See the discussion of the `-o` option in Section 5.2.14.3 on page 144.

`-g [`*group*`[,]][`*gid*`]` This option defines the group name and/or gid of the destination file. See the discussion of the `-g` option in Section 5.2.14.3 on page 144.

5.2.14.5 *Excluding Files*

`exclude` ***source***

A file listed after the **exclude** keyword that was previously included, for example from a recursive file definition, is excluded from the list of files.

If the source specifies a directory, then all files below that directory are excluded.

5.2.14.6 *Including Files*

`file <` ***include_file***

The **file** keyword can be used to include definitions for files from a separate include_file by specifying a < (less than) character followed by the include_file.

5.2.15 Space Control_file

path `[+|-]`*size*

For each path listed in the `space` file, the *swinstall* utility adds the size, in bytes, to the disk space requirements. The size can be positive (reflecting the maximum transient or permanent disk space required for the install), or negative (reflecting space freed by one of the scripts executed by the *swinstall* command). An implementation must consider positive records and may consider negative records.

5.3 Serial Format and Multiple Media

A distribution in the serial format of the software packaging layout is a bit-stream representation, implemented as a set of POSIX.1 extended *cpio* or extended *tar* archives which contain files in the directory structure of the software packaging layout defined in the beginning of this Chapter.

A serial distribution can be stored on any serial medium. A serial distribution can also be stored in any file, within the file system, which supports the storing of POSIX.1 extended *cpio* or extended *tar* archives. How a system reads or writes to the different media devices is outside the scope of this Software Administration specification.

Implementations support serial distributions if the underlying operating system supports the *pax* utility, as defined in POSIX.2, or otherwise supports reading and writing of the extended *tar* and extended *cpio* archives defined in POSIX.1. If serial distributions are supported, the serial distribution formats supported include extended *tar* and extended *cpio*.

The distribution is implemented as a set of one or more POSIX.1 extended *cpio* or extended *tar* archives. The archives reside on a set of one or more serial media, or in a file. Each media in a serial distribution contains one and only one archive.

A distribution may span multiple media in a hierarchical structure. In this case, the set of files on any particular media, including the attributes defined in any software definition files, should be similar to that for a serial archive. In other words, the decision for which files are put on which media should be the same whether the distribution is serial or hierarchical. Space considerations on media may cause some differences.

The following are the rules regarding ordering of files within serial distributions. These rules, including generation of the ***fileset.media_sequence_number***, are implemented by the ***swpackage*** utility.

1. The catalog files (which contain all the information describing the software contained in the distribution), as well as the control scripts, in this relative order:
    a. The global `INDEX` file, as described in Section 5.1.1 on page 126.
    b. The distribution files, as described in Section 5.1.1 on page 126.
    c. The product catalog files, product by product, as described in Section 5.1.1 on page 126.
        i. The product control files, as described in Section 5.1.1 on page 126.
        ii. The fileset control files, fileset by fileset, as described in Section 5.1.1 on page 126.
2. The actual software files, fileset by fileset, as described in Section 5.1.2 on page 128.
    a. Prerequisites of filesets before the filesets that depend on them
    b. Kernel filesets before non kernel filesets (except where kernel filesets have prerequisites on non-kernel)
3. Each medium has (as its first file, if a serial medium)
    a. A global `INDEX` file, `catalog/INDEX` that contains at least the ***distribution.uuid*** and ***media.sequence_number*** attributes (used to identify a particular media within a particular distribution)
4. Each archive starts at the beginning of the medium. Multiple archives on one medium are not allowed.

Additionally, in order to increase the usability of multiple media serial distributions, the following guidelines should be used and in decreasing importance:

- Each medium should contain complete files wherever possible. If a file is larger than the the capacity defined by the *media_capacity* option, then the behavior is implementation defined.
- Each medium should contain complete filesets whenever possible or practical. The *fileset.media_sequence_number* attribute is the number of the medium where the fileset begins. If a fileset is larger then the medium size, then the *fileset.media_sequence_number* attribute contains the list of *media.sequence_number*s describing the media that contain this fileset.
- Each medium should contain complete products whenever possible or practical.
- Each medium should contain needed dependencies whenever possible or practical.

Thus a conforming implementation is able to:

- Read the `INDEX` off of the first medium for the Selection Phase
- Scan the first medium (and those following as needed) for the necessary catalog files for the Analysis Phase
- Request the next needed medium for the next needed fileset based on *media_sequence_number* during the Execution Phase
- Request the next medium when the fileset spans media

Note that in all respects, a serial distribution conforms to the specifications of the extended *cpio* or extended *tar* archives. See POSIX.1. This includes, but is not limited to, the following:

- Recording format
- Character sets

Appendix A

Sample File Coding

A.1 Defaults File

The following is an example defaults file:

```
# File:        /var/adm/sw/defaults
# Description: This file contains example system defaults

  swinstall.allow_downdate                  =false
  swinstall.allow_incompatible              =false
  swinstall.ask                             =false
  swinstall.autoreboot                      =false
  swinstall.autoselect_dependencies         =true
  swinstall.defer_configure                 =false
  swinstall.distribution_source_directory   =/var/spool/sw
  swinstall.enforce_dependencies            =true
  swinstall.enforce_dsa                     =true
  swinstall.enforce_locatable               =true
  swinstall.enforce_scripts                 =true
  swinstall.installed_software_catalog      =/var/adm/sw/catalog
  swinstall.logfile                         =/var/adm/sw/swinstall.log
  swinstall.loglevel                        =1
  swinstall.reinstall                       =false
  swinstall.select_local                    =true
  swinstall.software                        =
  swinstall.targets                         =
  swinstall.verbose                         =1

  swcopy.autoselect_dependencies            =true
  swcopy.distribution_source_directory      =/var/spool/sw
  swcopy.distribution_target_directory      =/var/spool/sw
  swcopy.enforce_dependencies               =true
  swcopy.enforce_dsa                        =true
  swcopy.logfile                            =/var/adm/sw/swcopy.log
  swcopy.loglevel                           =1
  swcopy.recopy                             =false
  swcopy.select_local                       =true
  swcopy.software                           =
  swcopy.targets                            =
  swcopy.verbose                            =1

  swremove.autoselect_dependents            =false
  swremove.distribution_target_directory    =/var/spool/sw
  swremove.enforce_dependencies             =true
  swremove.enforce_scripts                  =true
  swremove.installed_software_catalog       =/var/adm/sw/catalog
  swremove.logfile                          =/var/adm/sw/swremove.log
  swremove.loglevel                         =1
```

```
swremove.select_local                     =true
swremove.software                         =
swremove.targets                          =
swremove.verbose                          =1

swconfig.allow_incompatible               =false
swconfig.allow_multiple_versions          =false
swconfig.ask                              =false
swconfig.autoselect_dependencies          =true
swconfig.autoselect_dependents            =false
swconfig.enforce_dependencies             =true
swconfig.installed_software_catalog       =/var/adm/sw/catalog
swconfig.logfile                          =/var/adm/sw/swconfig.log
swconfig.loglevel                         =1
swconfig.reconfigure                      =false
swconfig.select_local                     =true
swconfig.software                         =
swconfig.targets                          =
swconfig.verbose                          =1

swask.autoselect_dependencies             =true
swask.distribution_source_directory       =/var/spool/sw
swask.distribution_source_serial          =/dev/rmt/dat
swask.logfile                             =/var/adm/sw/swask.log
swask.loglevel                            =1
swask.software                            =
swask.targets                             =
swask.verbose                             =1

swmodify.distribution_target_directory    =/var/spool/sw
swmodify.installed_software_catalog       =/var/adm/sw/catalog
swmodify.files                            =
swmodify.logfile                          =/var/adm/sw/swmodify.log
swmodify.loglevel                         =1
swmodify.select_local                     =true
swmodify.software                         =
swmodify.targets                          =
swmodify.verbose                          =1

swverify.allow_incompatible               =false
swverify.autoselect_dependencies          =true
swverify.check_contents                   =true
swverify.check_permissions                =true
swverify.check_requisites                 =true
swverify.check_scripts                    =true
swverify.check_volatile                   =false
swverify.distribution_target_directory    =/var/spool/sw
swverify.enforce_dependencies             =true
swverify.enforce_locatable                =true
swverify.installed_software_catalog       =/var/adm/sw/catalog
swverify.logfile                          =/var/adm/sw/swverify.log
swverify.loglevel                         =1
swverify.select_local                     =true
swverify.software                         =
swverify.targets                          =
```

```
swverify.verbose                          =1

swlist.distribution_target_directory      =/var/spool/sw
swlist.installed_software_catalog         =/var/adm/sw/catalog
swlist.one_liner                          =revision title
swlist.select_local                       =true
swlist.software                           =
swlist.targets                            =

swpackage.distribution_target_directory   =/var/spool/sw
swpackage.distribution_target_serial      =/dev/rmt/dat
swpackage.enforce_dsa                     =true
swpackage.follow_symlinks                 =false
swpackage.logfile                         =/var/adm/sw/swpackage.log
swpackage.loglevel                        =1
swpackage.media_capacity                  =1330
swpackage.media_type                      =directory
swpackage.psf_source_file                 =psf
swpackage.software                        =
swpackage.verbose                         =1
```

A.2 Product Specification File

The following is an example product specification file:

```
# File:         psf.posix
# Description: This illustrates the structure of a typical application
#               software product.
distribution
# Vendor definition
    vendor
        tag              FineSoft
        title            Fine Software Corporation
         description     "Fine Software Corporation
                          1233 Technology Way
                          Sunset Bay, Ca, 90456
                          1-800-555-1231"
# Bundle definition:
    bundle
        tag              POSIX-SM
        title            POSIX 1387 System Management
        revision         1.0
        vendor_tag       FineSoft
        contents         POSIX-Printer,r=4.0  POSIX-Software,r=2.0
        contents         POSIX-User,r=1.0

# Product definition:
    product
        tag              POSIX-Software
        title            POSIX 1387.2 Software Administration Utilities
        revision         2.0
        vendor_tag       FineSoft
        number           J2326AA
        description      < /build/data/description
        copyright        < /build/data/copyright
        machine_type     9000/[78]*
        os_name          HP-UX
        os_release       ?.09.*
        os_version       ?
        directory        /

    # Subproduct definitions:
        subproduct
            tag          Manager
            title        management utilities
            contents     commands agent man
        subproduct
            tag          Agent
            title        target daemon and agent
            contents     agent man

    # Fileset definitions:
        fileset
            tag          commands
            title        Commands (management utilities)
            configure    scripts/configure.data
            prerequisite POSIX-Software.agent
            directory    /build/usr/sbin  /usr/sbin/
            file         swinstall
            file         swconfig
```

```
    file         swcopy
    file         swlist
    file         swremove
    file         swverify
    file         swpackage


fileset
    tag          agent
    title        Agent (target agent)
    configure    /build/system/SD-AGENT/customize
    unconfigure  /build/system/SD-AGENT/decustomize
    file         /build/usr/sbin/swagentd  /usr/sbin/swagentd
    file         /build/usr/lbin/swagent  /usr/lbin/swagent

fileset
    tag          man
    title        Manual (man pages)
    directory    /build/usr/man/man8  /usr/man/man8
    file         swinstall.8
    file         swcopy.8
    file         swremove.8
    file         swlist.8
    file         swverify.8
    file         swconfig.8
    file         swpackage.8
    file         swagent.8
    file         swagentd.8
    directory    /build/usr/man/man4  /usr/man/man4
    file         swpackage.4
```

A.3 Software Packaging Layout

The following is an example software packaging layout created from the PSF in Section A.2 on page 152, for a distribution located at `/var/spool/sw/`:

```
catalog
catalog/INDEX
catalog/dfiles
catalog/dfiles/INDEX
catalog/dfiles/INFO
catalog/POSIX-Software
catalog/POSIX-Software/pfiles
catalog/POSIX-Software/pfiles/INFO
catalog/POSIX-Software/agent
catalog/POSIX-Software/agent/INFO
catalog/POSIX-Software/agent/configure
catalog/POSIX-Software/agent/unconfigure
catalog/POSIX-Software/commands
catalog/POSIX-Software/commands/INFO
catalog/POSIX-Software/commands/configure
catalog/POSIX-Software/man
catalog/POSIX-Software/man/INFO
POSIX-Software
POSIX-Software/agent
POSIX-Software/agent/usr
POSIX-Software/agent/usr/lbin
POSIX-Software/agent/usr/lbin/swagent
POSIX-Software/agent/usr/sbin
POSIX-Software/agent/usr/sbin/swagentd
POSIX-Software/commands
POSIX-Software/commands/usr
POSIX-Software/commands/usr/sbin
POSIX-Software/commands/usr/sbin/swinstall
POSIX-Software/commands/usr/sbin/swpackage
POSIX-Software/man
POSIX-Software/man/usr
POSIX-Software/man/usr/man
POSIX-Software/man/usr/man/man4
POSIX-Software/man/usr/man/man4/swpackage.4
POSIX-Software/man/usr/man/man8
POSIX-Software/man/usr/man/man8/swagent.8
POSIX-Software/man/usr/man/man8/swagentd.8
POSIX-Software/man/usr/man/man8/swconfig.8
POSIX-Software/man/usr/man/man8/swcopy.8
POSIX-Software/man/usr/man/man8/swinstall.8
POSIX-Software/man/usr/man/man8/swlist.8
POSIX-Software/man/usr/man/man8/swpackage.8
POSIX-Software/man/usr/man/man8/swremove.8
POSIX-Software/man/usr/man/man8/swverify.8
```

A.4 INDEX File

The following is an example **INDEX** file for the software packaging layout in Section A.3 on page 154.

```
distribution
    layout_version 1.0
    uuid 944B41Z-X135
media
    sequence_number 1
vendor
    tag FineSoft
    title Fine Software Corporation
    description "Fine Software Corporation
                 1233 Technology Way
                 Sunset Bay, Ca, 90456
                 1-800-555-1231"
bundle
    tag POSIX-SM
    title POSIX 1387 System Management
    revision 1.0
    vendor_tag FineSoft
    contents POSIX-Printer,r=4.0 POSIX-Software,r=2.0 POSIX-User,r=1.0
product
    tag POSIX-Software
    instance_id 1
    control_directory POSIX-Software
    revision 2.0
    vendor_tag FineSoft
    title POSIX 1387.2 Software Administration Utilities
    description "The objective of this standard is to address this
    problem for software administration, a specific area of system
    administration, and to contribute to the overall solution of
    administering computing environments, both stand-alone and
    distributed.

    In pursuit of this goal, this standard defines a set of utilities,
    a set of objects acted upon by those utilities, a set of information
    maintained about installed software, and the layout on a physical
    medium of software awaiting installation.
    These definitions provide the flexibility necessary for system
    administrators to enforce policies suitable to their environments."
    directory /
    machine_type 9000/[78]
    os_name HP-UX
    os_release ?.09.
    os_version ?
    all_filesets agent commands man
    number J2326AA
    copyright <copyright
subproduct
    tag Agent
    title target daemon and agent
    contents agent man
subproduct
    tag Manager
    title management utilities
    contents commands agent man
fileset
```

```
        tag agent
        control_directory agent
        size 5333089
        title Agent (target agent)
        state available
    fileset
        tag commands
        control_directory commands
        size 8531074
        title Commands (management utilities)
        state available
        prerequisite POSIX-Software.agent
    fileset
        tag man
        control_directory man
        size 162749
        title Manual (man pages)
        state available
```

A.5 INFO File

The following is an example **INFO** file for the software packaging layout in Section A.3 on page 154, and the **INDEX** file in Section A.4 on page 155.

```
control_file
    path INFO
    size 638
    tag INFO
control_file
    path configure
    size 3023
    tag configure
control_file
    path unconfigure
    size 375
    tag unconfigure
file
    path /usr/lbin/swagent
    type f
    size 2973696
    cksum 3139283961
    mode 0555
    uid 0
    gid 3
    owner root
    group sys
    mtime 739080771
file
    path /usr/sbin/swagentd
    type f
    size 2355200
    cksum 1313249400
    mode 0555
    uid 0
    gid 3
    owner root
    group sys
    mtime 739081332
```

A.6 Control Script

The following is an example control script for the software packaging layout in Section A.3 on page 154, the **INDEX** file in Section A.4 on page 155, and the **INFO** file in Section A.5 on page 157.

```
#
# agent configure script
#
PATH=$SW_PATH
ROOT=$SW_ROOT_DIRECTORY
BASE_DIR=$ROOT/$SW_LOCATION
MY_CATALOG=$SW_ID
MY_CONTROL_DIR=$SW_CONTROL_DIRECTORY
MY_SCRIPT=$SW_CONTROL_TAG
OPTIONS=$SW_SESSION_OPTIONS

# Make sure /var/adm/sw/ exists
if [ ! -d $BASE_DIR/var/adm/sw ]
then
    mkdir -p $BASE_DIR/var/adm/sw
    chmod 555 $BASE_DIR/var/adm/sw
fi

# Make sure we are running on "/" (swconfig should enforce this)
if [ $ROOT != "/" ]
then
    # Exit error
    echo "ERROR:   Trying to run $MY_SCRIPT when root is not /."
    exit 1
fi

# Kill and restart the daemon
daemon_process="`ps -e|grep swagentd|grep -v grep|awk '{print $1}'`"
if [ "${daemon_process}" != "" ]
then
    kill $daemon_process
    if [ $? != 0 ]
    then
        # Exit warning
        echo "WARNING: Can not kill and restart the daemon."
        exit 2
    fi
fi
/usr/sbin/swagentd

# Exit success
exit 0
```

A.7 Patch PSF Example

```
product
    tag             OS-Core
    revision        B.10.01.006
    architecture    HP-UX_B.10.01_700
    vendor_tag      HP

    title           Core Operating System (patch)

    machine_type    9000/7??
    os_name         HP-UX
    os_release      ?.10.0*
    os_version      *

    is_patch        true

    category_tag    critical


    fileset
        tag CMDS-MIN
        title "Patch of csh and who"
        description " Patch of csh and who ... blah blah blah."

        # assume that other patches to base fileset used up 004 and 005
        revision B.10.01.006

        ancestor OS-Core.CMDS-MIN,r=B.10.01,a=HP-UX_B.10.01_700,v=hp

        is_patch     true      # this is a patch/sparse fileset
        supersedes OS-Core.CMDS-MIN,r=B.10.01.002
        supersedes OS-Core.CMDS-MIN,r=B.10.01.003

        file /build2/usr/bin/csh   /usr/bin/csh
        file /build/sbin/who       /sbin/who
    end
end
```

Appendix B

Background Information

B.1 General

B.1.1 Scope and Purpose

This Appendix provides background information on the approach adopted by the developers of the 1387.2 Standard, and the reasons why it was specified in the way it is. It is hoped this information will be helpful to both implementors and users.

A number of areas are not covered in the 1387.2 Standard. A few things (such as physical media) are truly outside the scope of the Standard. However, some things are listed as either undefined or unspecified.

The requirements for all POSIX.2 utilities and all the file system features of POSIX.1 were significant issues in formulating this Standard. It was asserted that the underlying operating system need not be fully POSIX.1 or POSIX.2 conformant. With the requirement left as it is, implementation on such systems as DOS, OS/2, MVS, VMS, etc., is feasible.

Implementers of this Standard should provide some guidance to those who write scripts which will be packaged in distributions. It is actually those scripts that have significant dependencies on the features of the underlying operating system. Assured portability of scripts is not possible without assurance of an interpreter and utilities. By allowing other interpreters, some concession has been made to assist in managing existing software in the real world. The best portability assumption is that the `checkinstall`, `preinstall`, and `postinstall` scripts should not depend on features beyond those of POSIX.2 or POSIX.1. The `configure` scripts run only on the systems that actually use the software, and hence they need not be as portable as the `preinstall` and `postinstall` scripts.

This Standard specifies distributed operations without specifying the mechanism for how it is to be achieved.

Work to specify interoperability for this Software Administration specification has been taken up by The Open Group, which has published a specification defining interoperability using the Distributed Computing Environment (DCE) remote procedure calls (RPC) — see reference **XDSA-DCE**. Specifications for other technologies to provide distributed XDSA working will be added as and when industry support for them becomes evident.

B.1.2 Roles

In defining standards for software administration, the concept of roles is used to specify the way interactions occur in order for software administration to take place. Figure B-1 shows the various roles, including those that are outside the scope of this Software Administration specification, as well as those within that scope.

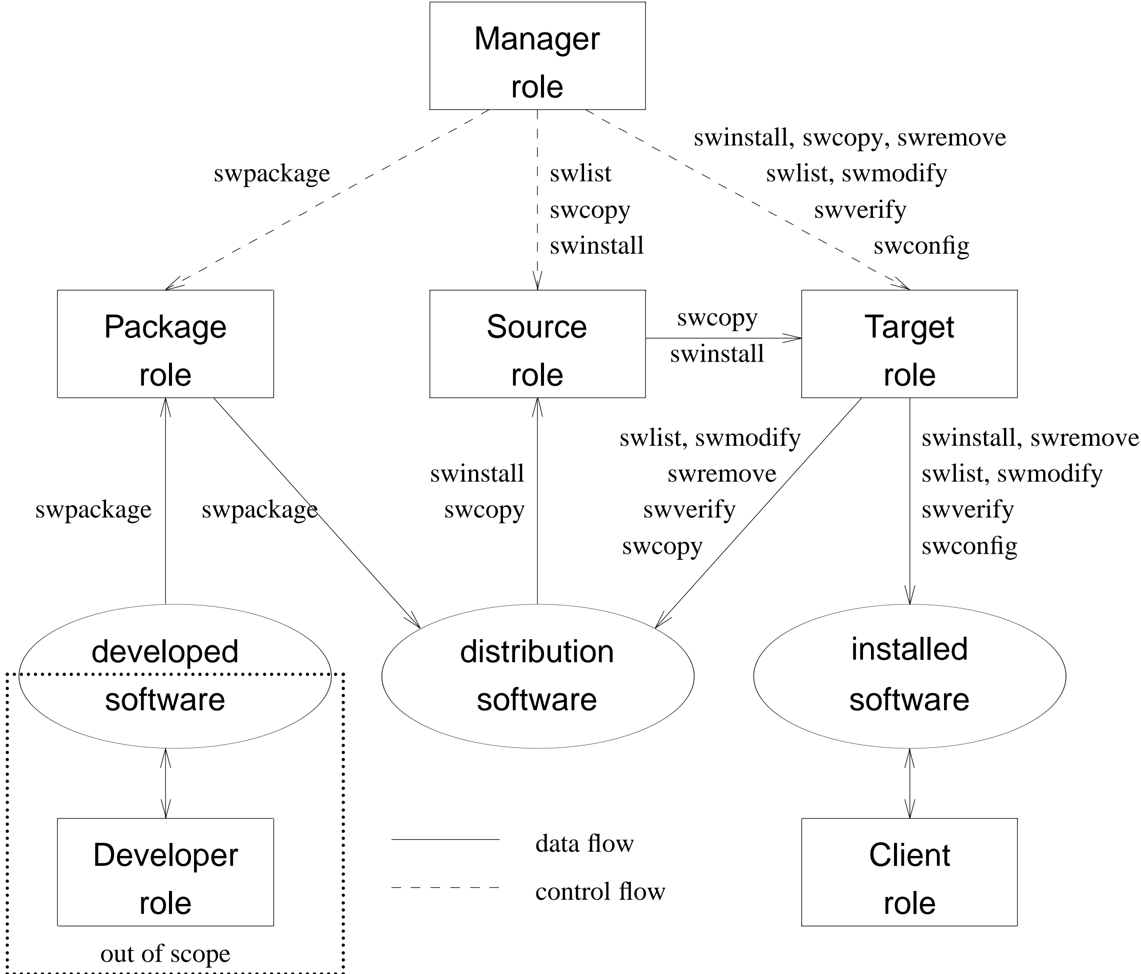

Figure B-1 Roles in Software Administration

Distributed applications require actions to be performed in more than one place (system or directory). These distributed portions have often been referred to as client and server. Software administration tasks also are often initiated by different users at different times. Since the terms client and server have implementation implications beyond the scope of this Software Administration specification, the more neutral concept of role is introduced. This stems from a need to refer to things that occur logically, if not physically, on what might be thought of as a client or server. In the context of this Software Administration specification, roles are simply a convenient way of referring to where function is apparent, with no implication for how this is actually implemented.

It may be helpful to think of roles as separate processes, one per role, but that is only one possible implementation. Roles may operate on separate systems, or hosts, although all roles may operate on the same host. For example, the packager role creates an initial distribution. When copying this distribution, the source role provides read access to the distribution files, while the target role writes the new copy. This new copy may then be read by another source role for another install or copy.

For any implementation, a role consists of the entire set of tasks that may occur within the role. A task is a set of well-defined behaviors and state changes in the managed objects. Tasks are

initiated by the system administrator using a specific command in the command line interface (CLI). Tasks are defined by this Software Administration specification in terms of the state changes on the software objects on target hosts.

As each task proceeds, different roles are involved. These roles may be realized on a single machine or could involve a different machine for each role.

Developer Role

Where the software is developed, tested, and maintained.

This role is outside the scope of this Software Administration specification. In Figure B-1 on page 162, software is developed by the development role in some environment that results in it being in the developed state.

Manager Role

Where each task is initiated and is concerned with taking appropriate action at the completion or failure of a task.

Manager control is understood as a more common need than target control, so at least that should be supported. For this reason, the manager role sets the options for a task, and each of the target hosts implements those options. So, any extension involves a set of ways to define selected control over particular policies. A design for this has not been pursued beyond recognizing the complexity of the problem.

The manager role provides the means of controlling the way software is created, transferred, and installed. In particular it provides an administrative interface to the other roles, enabling their activities to be controlled in a coordinated manner.

Packager Role

Where software that has been developed is organized in a form suitable for distribution.

The packager role transforms a product from the format produced by the developer role to the format specified by this Software Administration specification for use by the next stage of the process, the source, and manager roles. The packager role defines the requirements for this transformation to be successful the input (the product specification file, and the files it describes), the command line interface to initiate the transformation (*swpackage*) utility, and the output of the packaging task (packaging layout).

Two distinct, but related, formats for packaged software are supported by this Software Administration specification a structured format residing within a POSIX.1 hierarchical file system (such as disk, CD, etc.), and a bit stream representation residing on any serial device or file (such as tape, *tar* archive file, etc.).

Source Role

Where the software exists in a form suitable for distribution and hence forms a context for the establishment of a repository of software from which the manager may choose to distribute to the target.

Software exists in the source until it is removed by a task initiated by the manager.

The source role provides a repository where software may be stored, and provides access for those roles that require the software.

Target Role

The target of a task.

For example, when installing software, the target is where software is installed after having been delivered from a source. As another example, the target for a copy task command refers to the distribution where products are added. For management tasks like removing

software, the target refers to either the installed_software objects or the distributions from which software is being removed.

Part of the distributed model involves the target role granting permission to the manager role to perform various software administration tasks. Authority for certain classes of tasks may be individually controllable, for example, modifying vs. listing installed products. While it is entirely conceivable that the target may want to restrict the way authorized tasks are performed, it is beyond the scope of this Software Administration specification.

Client Role

Where the software is actually executed or used, which may be different from where it is actually installed.

Software is configured for use on the client.

An example is installing for an environment where many hosts share software from one system. Diskless systems are one example of systems that do such sharing. A manager role initiates the install task with the source role serving the software from the distribution and the target role installing the software on a fileserver. After the installation is complete, then a client role on each client sharing this software performs configuration for the shared software and the client host.

It is important to understand the difference between the target and client roles. The client role is where the software is actually used and where configuration of the software takes place, while the target role is where the software is installed. Although in many cases these are the same machines, in some cases they are different and the separation of configuration from installation is important. Each target from the *targets* operand of an install or configure task may identify a target role (if installing but not configuring), a client role (if just configuring), or both (if installing and configuring).

Two examples of when these roles refer to different machines are:

- Proxy install (installing on one system for use by another system) where configuration of the software is done separately, because the target may not have the necessary capabilities or information, or both, for configuring the client.
- The target is a file server, and there are multiple clients that access the software installed on the file server. Each client may require separate configuration as targets of *swconfig*.

Figure B-1 on page 162 shows a split between the manager role and the other roles. The administrative interface to software administration is provided in the manager role, from which the individual tasks that take place in the other roles are controlled.

This Software Administration specification defines a set of utilities that is such an administrative interface. These utilities provide basic facilities for controlling the individual tasks. Other management applications may be built that provide much more comprehensive software administration facilities. This Software Administration specification defines facilities that enable management applications to control software administration across any number of systems with conforming implementations.

One item of note among the general terms is the definition of symbolic link (see the Glossary). While not yet standardized (see referenced document **B21**), symbolic links are an entrenched part of existing practice. This Software Administration specification makes no attempt to independently define symbolic links. Rather, the functional characteristics of symbolic links are undefined.

B.1.3 Tasks

Software administration involves the control of software throughout the software life cycle from the organization or creation of a software object through its installation, maintenance phases, and eventual removal.

The following tasks are identified in this Software Administration specification. The defined utilities provide a way of accomplishing these tasks except as noted.

Install software (`swinstall`)

This task takes software from a source distribution and installs it on a target file system in a form suitable to be configured on this system or another system sharing this software. Parts of software products (subproducts or filesets) can be installed or reinstalled at different times.

In the case where the system on which the software is installed will also be using the software (that is, it is acting as both a target and client role), configuring the software can be combined with the install software task.

Reinstall software (`swinstall`)

This task is simply installing the exact same software that was previously already installed.

Configure software (`swconfig`)

This task takes place on the client role that will be using the installed software. Configuration makes that software ready to use. Configured software can also be reconfigured as required or can be unconfigured (to deactivate a particular version or prepare it for removal).

Update software (`swinstall`)

This task updates the target file system by installing a newer revision of software than is already installed. This is also referred to as upgrading.

The new revision of software can be installed in the same location as the current revision. In this case, the software `configure` scripts executed by the configure task need to handle saving or updating the necessary configuration data.

The new revision of software can alternatively be installed in a location different than the current revision. In this case, the old revision may be unconfigured by the `unconfigure` script executed as part of the unconfigure task, and the new revision is configured by the `configure` scripts executed as part of the configure task.

Downdate software (`swinstall`)

This task ‘‘downdates’’ the target file system by installing an older revision of software than is already installed. This is also referred to as ‘‘downgrading’’ or ‘‘reverting.’’

The older revision of software can be installed in the same location as the current revision. In this case, the configuration process of the older version handles the necessary changes in configuration.

The older revision of software can alternatively be installed in a location different from that of the current revision. In this case, the new revision can be unconfigured via the unconfigure task, and the older revision can be configured either independently, or as part of install.

Recover software (`swinstall`)

This task restores the previous version of software (if it exists) in the case where an update, downdate, or reinstall of software fails. This Software Administration specification defines the minimum required support for automatic recovery process in the install task.

Apply software patch (`swinstall`)
: This task replaces part of a software fileset with a new set of files by installing a fileset with those new files in the same location as the fileset being patched. This is also referred to as fixing software.

  Updates and patches can be implemented through standard 1387.2 facilities and control scripts, although these control scripts can become quite complex. This Software Administration specification also proposes significant enhancements to the standard to facilitate these operations.

Remove installed software (`swremove`)
: This task removes software from an installed_software object where it previously was installed. Parts of software products (subproducts or filesets) can be removed at different times.

  If the system where the software is installed was also using the software, unconfiguring the software can be combined with the remove software task.

Remove software patch (`swremove`)
: This task removes a patch fileset. This is also referred to as rejecting software.

  Filesets related through naming conventions and prerequisites can be used. Restoring patch files when removing the patched can be achieved via remove control scripts.

Verify the installed software (`swverify`)
: This task checks that software previously installed still exists and is intact. If operating on a system that was configured to use the software, it can also check that the software is configured properly.

List installed software information (`swlist`)
: This task provides a list of the software that has been installed on a target. Options are available to specify which software packages are to be listed and to control the amount of information provided.

Fix installed software information (`swmodify, swverify`)
: This task modifies information about software that has been installed on a target. Options are available to specify which software packages, and what information about those packages, are modified.

Package software (`swpackage`)
: This task takes place in the packager role and transforms developed software into the software packaging layout suitable for distribution. The metadata that defines the software objects to be packaged is contained in the product specification file (PSF).

Copy distribution software (`swcopy`)
: This task copies distribution software between a source and a target, for subsequent use of that target as a source. Copying software can be used to merge distributions, to distribute products to the installation targets, and then install from that local copy, or to copy part of a distribution to a removable media for physical distribution (as opposed to electronic distribution).

Remove distribution software (`swremove`)
: This task removes products from a target distribution.

Check/verify distribution software (`swverify`)
: This task checks that a target distribution exists and is intact.

List distribution information (`swlist`)
: This task lists source or target distribution information. Options are available to specify

which objects in a distribution are to be listed, and to control the amount of information provided.

Fix distribution information (`swmodify, swverify`)
This task modifies information that describes, and is contained within, target distributions. Options are available to specify which objects in a distribution are modified.

License installed software (undefined)
How software licenses are managed is undefined within this Software Administration specification.

The task definitions were based on study of existing practice for software administration. This included presentations on existing practice by many different system vendors and system administrators. From these, a functionally adequate base was selected upon which all parties could build. While it was recognized that this did not address every concern, it was felt that that the utility descriptions (including detailed behavior), software structure definitions, and media layout, provided an excellent starting point. After comparing various existing practices, these choices appeared to be quite similar to other existing practices in many details of these key areas.

B.1.4 Update Requirements

This XDSA specification enhances the IEEE 1387.2 Standard in its provision for handling updates.

Customer update requirements for software updates were perceived as follows:

- Ability to update a previous release with a new release with the same reliability, performance and resultant functionality as installing the new release for the first time, at the same time preserving any customer data or configuration information.
- Automatic selection of the most recent release, or manual selection of previous releases.
- For network distribution, do not necessarily redistribute files from the new release that have not changed.
- Record that the previous release has been updated (that is, no longer exists) in the installed software database.
- As an alternative to in-place updates, support multiple releases of software on a system, letting the user choose which one to use.
- Goal of no down time while upgrading.
- Support automatic rollback to the previous release if the installation fails.
- Ability to store multiple releases of multiple architectures of software in the same depot.
- Easy identification of different releases of the same software.

Provider update requirements for software updates were perceived as follows:

- Ability to easily repackage and redeliver a software update.
- Ability to have a single package that supports both new installs and update of previous installs.
- Easy management of customer data and configuration information.
- Ability to deliver multiple releases of software in the same distribution.
- Provide for automatic removal of obsolete files from the release that has been updated.

- Ability to have a ''sparse'' update, where only a subset of the files in a fileset is updated, but the result is the same as a full update.

B.1.5 Patch Requirements

This XDSA specification enhances the IEEE 1387.2 Standard in its provision for handling patches.

Customer patch requirements were perceived as follows:

- Ability to store patches in the same depot as other software.
- Easy identification of available or installed patches and of what filesets have been or can be patched.
- Automatic installation and copying of patches along with their base filesets.
  - Allows installation/copying of patches in the same session as the base software, or during cold install.
- Improve patch installation performance over script based patches.
- Make patch selection easier by allowing selection of patches based on criteria (categories).
  - Criteria could cover categories such as severity, quality, and special cases such as hardware enablement.
- Provide for automatic removal of patches when software is updated, downdated, reinstalled or removed.
- Allow automatic selection of patches matching the installed software on target.
  - Installs/copies patches matching software installed on target system (or in target depot).
  - Criteria categories provide additional control over selection.
- Allow rollback of patches and automatic restoration of previous versions of patched files.
  - Switchable rollback capability.
  - No special scripts required.
- Provide a mechanism for patch commit. ''Commit'' of a patch removes rollback information (recovers disk space used to save previous file versions).
- Ability to list what files will be updated.
- Ability to find out what problems will be fixed by this patch.

Patch provider and support organization requirements were:

- Simplified native patch structure, improved reliability.
- Faster patch production.
- Ability to deliver patches on a single distribution media along with base software.
- Integrate ''reasonably'' into current patch processes; support both point patches and new patches superseding one or more previous patches; support patching of libraries.
- Support patching of multiple releases.

B.1.6 Conformance

The intended conformance classes defined in this Software Administration specification are derived somewhat from the examples of POSIX.1 and POSIX.2, with variations to support unique situations.

Implementation conformance is intended to be based on implementation of the utilities defined in this Software Administration specification, and on the proper POSIX.1 and POSIX.2 support from the operating system.

There is scope for a new conformance class — Distribution Conformance — to allow suppliers of software to package their software in a conformant manner. Distributions have many of the characteristics of applications using POSIX.2, since the distributions contain executables (presumably shell scripts).

B.1.6.1 Implementation Conformance

This class of conformance would require support of all the POSIX.1 and POSIX.2 functionality referenced in this Software Administration specification. The requirements from POSIX.1 are primarily for hierarchical file system support, including the file attributes of owner, group, and mode. In addition, the POSIX.2 utilities are required to support portable scripts.

This would assure that every Conforming Implementation would be able to install any strictly conforming distribution properly, including the proper settings of file attributes. One might question this need if one is installing software particular to a system that is not POSIX.1 conformant. It is the pervasive ability to serve the software over a distributed file system that makes critical the need for all conforming implementations to understand at least one set of well specified operating system behavior. The one set of operating system behavior chosen is POSIX.1. The need for POSIX.2 is primarily driven by the presence of executable control files within distributions. At least one guaranteed mechanism is required to invoke those files, and the shell interpreter was chosen for that purpose. Further, developers of portable scripts need a guarantee of some basic set of utilities with which to work, and the POSIX.2 utilities were chosen for that purpose.

A conforming implementation need not include the POSIX.1 and POSIX.2 implementation itself, but it must document how such can be obtained for the systems that the implementation supports. It is reasonable to assume that a given implementation, conformant in the presence of proper POSIX.1 and POSIX.2 support from the operating system, may still operate correctly on some distributions even when the proper operating system support is not present in full or in part.

B.1.6.2 Distribution Conformance

Strictly Conforming POSIX.2 Distribution

The Strictly Conforming Distribution class is intended to provide the highest degree of portability for a distribution. Conformance to such a class would guarantee that any conforming implementation could install this software properly.

Conforming POSIX.2 Distribution

The Conforming Distribution class is intended to guarantee that any conforming implementation can copy or install this software properly. This class would also allow for additional functionality, which may come either from implementations that can take advantage of additional attributes, or from software being able to store and retrieve that information from any Conforming Implementation.

Conforming POSIX.2 Distribution Using Extensions

This class is intended to allow evolution of this Standard, but in an open, consistent and well-documented manner. Examples of this are compressed media or bootable serial media. Both of these are features were recognized by the developers of this Standard as important, but consensus to include them in the 1387.2 Standard was not achieved.

This class would also provide flexibility for distributions needing to conform to other constraints related to the support of POSIX.1 and POSIX.2. There was strong support among the user community in the 1387.2 development project for support for interpreters other than *sh*. Support for other interpreters also permits the use of such distributions on systems, such as DOS, which are not conformant with POSIX.1 or POSIX.2.

B.2 Software Structures

B.2.1 Classes and Attributes

An example of the structure of the software objects for this Software Administration specification is illustrated in Figure B-2.

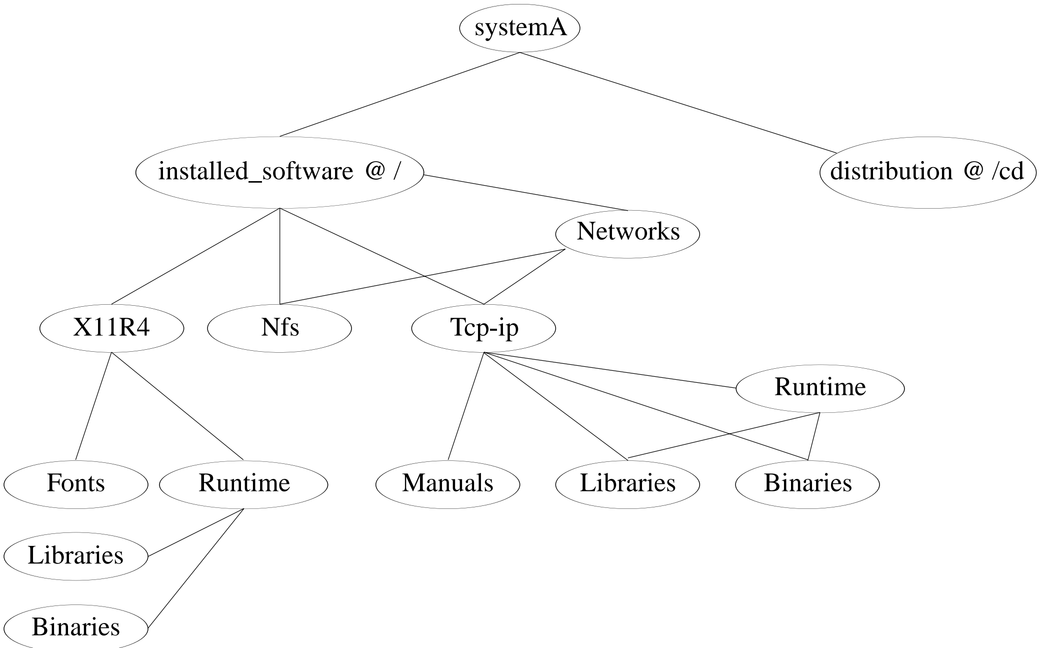

Figure B-2 Example of Software Structure

At the top of the hierarchy is a host, which is a system that conforms to this Software Administration specification. It is the starting point for finding all the software on that system that falls within this Software Administration specification. A host contains software_collections.

There are two distinct types of software_collections, as listed in the following, that may exist within a conformant system:

distribution
: A distribution consists of software products, in a form ready for installation. A distribution may also contain software bundles. There may be many distributions within a host.

installed_software
: An installed_software object consists of products installed from a distribution. An installed_software object may also contain software bundles. There may be other installed_software objects for use by this system or for other systems.

Software is organized into a hierarchy of objects, as described in the following, that are operated on by the utilities defined in this Software Administration specification:

product
: A product consists of filesets and control scripts, plus all the associated metadata. The content of a product may be specified as a collection of subproducts, filesets, or a combination of the two.

bundle

A bundle is a grouping of other software objects and is a convenient way to reference a set of software.

fileset

A fileset consists of the actual files plus control scripts. Filesets are generally the lowest level of software object that can be operated on by the utilities.

subproduct

Subproducts are a grouping of other subproducts, or of filesets, or of some combination, that resolve to a group of filesets. Subproducts are a convenient way to aggregate filesets.

The software_files define the files and control_files that are contained in the software objects that are operated on during a software administration utility. There are two classes of software_files as described in the following:

control_file

Control_files consist of control scripts and other files that are used in various ways by the utilities. Control scripts are executed by the utilities at various points in a task. Control scripts provide a way to perform steps, in addition to those executed by the utilities, at various points in the task such as preinstall checking, postinstall customization, configuration, and verification. Either a single script with multiple entry points, or multiple scripts can be defined.

Most control scripts are run on the target, which may be a different architecture than the client on which the software operates. They should, therefore, use POSIX.2 utilities, except where they can determine that they are running on the client.

In addition to scripts, other control_files provide input to the control scripts, or to the utilities directly (for example, the `response` and `space` control_files).

file

Files are the lowest level of object defined by this Software Administration specification. Files contain the attributes describing the file including the contents of the file and its installed location.

The distributions and installed_software objects are the sources or targets of a software administration command. The software objects (products, filesets, bundles, and subproducts) are the objects that are being applied to those targets.

This Software Administration specification describes the structure and the attributes for software_collections, software objects, and software_files. It also describes the behaviors for the utilities that operate on these objects. However, these structure definitions are not managed object classes in the ISO sense because the behaviors are not described in terms of methods within object classes[17].

Figure B-3 on page 173 shows the components of the software object hierarchy. The containment arrows designate objects that are defined within the context of their containing objects. An object can only exist within one containing object. The identifier of an object (for

17. Object classes are templates for the creation of object instances. They are analogous to the definition statements used in programming languages to define data structures that will be created later. Objects contain more than data structures, in that they also possess methods (procedures that are executed by objects). A well-formed object class has methods defined that handle all object data manipulation, including creation, modification, and listing, so that the actual storage of the data is appropriately hidden from the application using the objects.

example, the *tag* attribute of a fileset) only needs to be unique within the scope of the containing object.

The reference arrows designate objects that are included when this object is operated on. An object may be referred to by more than one object. Bundles need not refer to entire products, but can refer to individual filesets or subproducts. Fileset and subproduct objects can be referenced directly by bundles by also identifying the product of which the fileset or subproduct is a part.

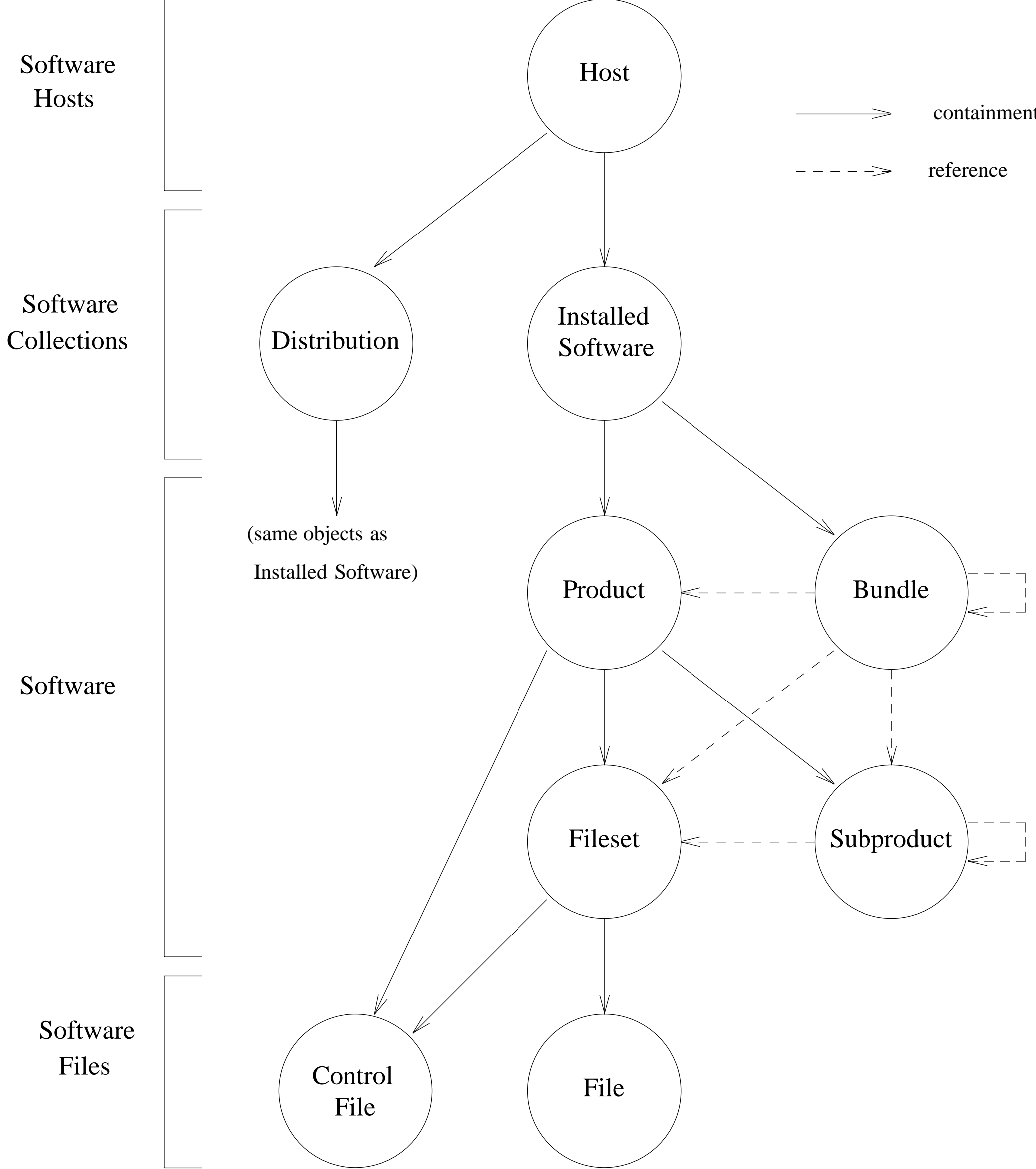

Figure B-3 Software Object Containment

Figure B-4 on page 174 shows the software administration common classes and the software objects that inherit attributes from these common classes.

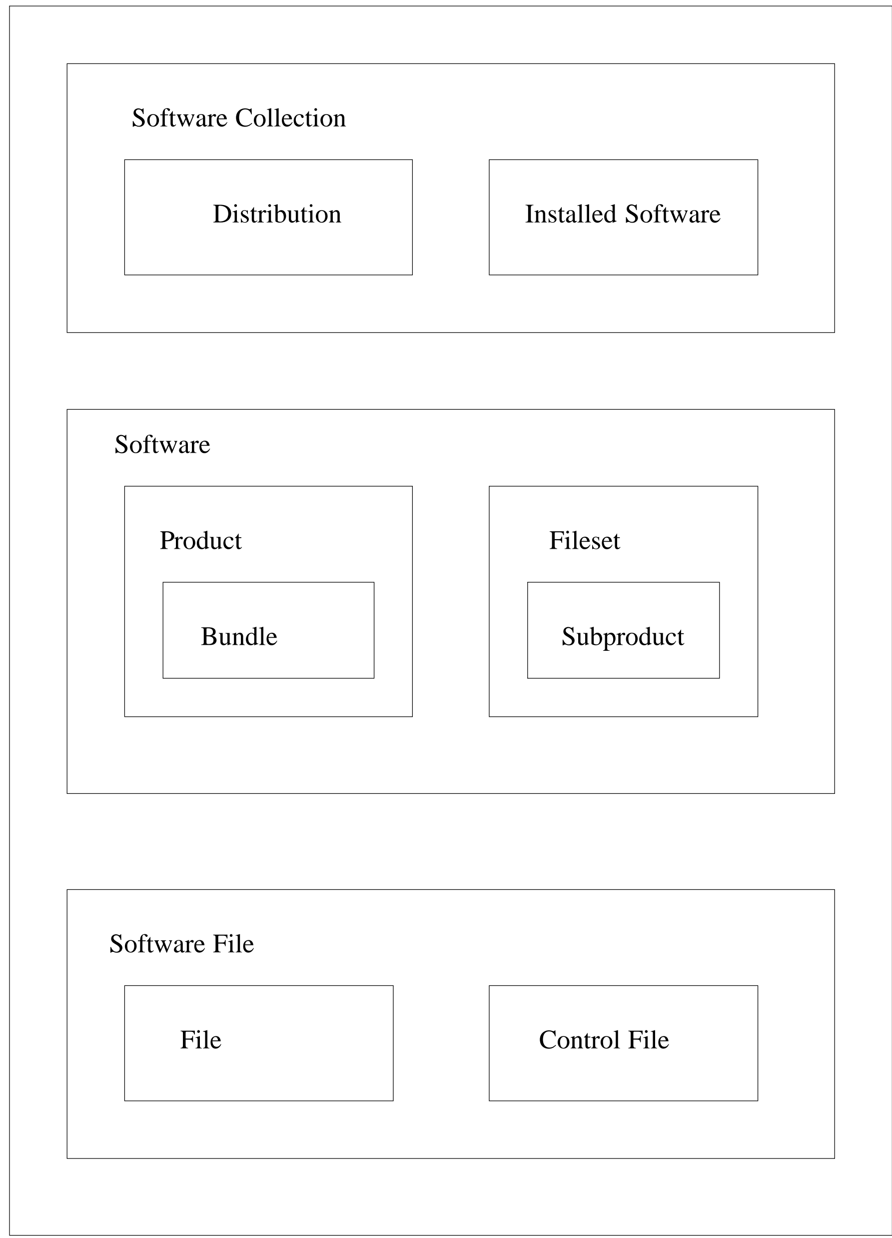

Figure B-4 Software Object Inheritance

Interoperability between implementations of this Software Administration specification may be achieved through the definition of methods for the first two of these common classes, "software_collections" and software. The software_collections are the source and target objects for software administration, while the software objects are the objects that are operated on within the context of the software_collections. Operations on individual software_files independent of operations on software objects is undefined.

This Software Administration specification also does not define how remote file systems are managed. In the simplest case, each file system is ''local'' to a single host, and all installations may be directed to the file system through an agent process on that host. Thus, files on a file system are contained within one of the installed software collections contained below that host.

On the other hand, an implementation may also choose to allow installation to a remote file system over a remote file system protocol. That is, the target process is running on a host that is different from the one that contains the file system. In this case, the files on that file system may be contained within the same software collection as before, or may be contained within a local software collection. In another implementation, all software collections may be stored within a global naming service instead of below any particular host.

An implementation may choose to define a software host object, or manage software as part of a more general host object. The attributes of a host object that are of interest to this Software Administration specification are shown in Table B-1.

Table B-1 Possible Attributes of a Host Class

Attribute	Length	Permitted Values	Default Value
host	Undefined	Portable character set	None
os_name	32	Portable character set	None
os_release	32	Portable character set	None
os_version	32	Portable character set	None
machine_type	32	Portable character set	None
distributions	Undefined	List of distribution directories	Empty list
installed_software	Undefined	List of installed_software directories and catalog identifiers	Empty list

The following are the attributes of the hosts that contain software_collections managed by this Software Administration specification:

distributions

The list of *distribution.path* attributes for distributions in the software host object.

These describe the `PATHNAME` portion of a software_collection source or target.

host

Identifier used to specify the host portion of a software source or target.

Identification of a remote host system is dependent on the networking services implementation and thus the syntax and semantics of the host name is undefined within this Software Administration specification.

installed_software

The list of *installed_software*.path and *installed_software*.catalog attributes for installed_software objects in the software host object.

These describe the `PATHNAME` portion of a software_collection target.

machine_type

Corresponds to the *machine* member of the *uname*() structure defined in POSIX.1 section 4.4.1.

It is the hardware type on which the system is running.

os_name

Corresponds to the *sysname* member of the *uname*() structure defined in POSIX.1 section 4.4.1.

It is the name of this implementation of the operating system.

os_release

Corresponds to the *release* member of the *uname*() structure defined in POSIX.1 section 4.4.1.

It is the release level of the operating system implementation.

os_version

Corresponds to the *version* member of the *uname*() structure defined in POSIX.1 section 4.4.1.

It is the version level of this release of the operating system.

B.2.2 Software_Collection

This class definition exists for convenience in defining the classes that inherit from it. It is not intended that any direct instances of this class be created, but only of the classes that inherit from it.

Multiple versions of products and bundles are possible when subsequent releases of a product or bundle have different revision numbers, and when products or bundles targeted for different machine types or other OS attributes define the architecture attribute differently.

The *layout_version* attribute is the version number of this Software Administration specification to which the distribution conforms. The name of this Software Administration specification (for example, P1387.2-19xx) was considered but there was concern that the delay between IEEE acceptance and ISO acceptance would make it hard to pick the year correctly. It is not clear when to change the number from 1.0 to 1.1 or even from 1.x to 2.0.

It is possible for an `INDEX` file describing a distribution to contain products with different values of *layout_version*. The software_collection *layout_version* refers only to the format of the distribution attributes and the **product** keyword. After the **product** keyword, the product *layout_version* defines the format of the definitions of all objects within that product.

B.2.3 Distribution

POSIX.1 allows for different pathname and filename sizes. Thus it is possible for a distribution to be created on one system and not be readable or installed on another system (each of which conforms with this Software Administration specification) because of differences in their POSIX.1 {NAME_MAX} and {PATH_MAX}. Consideration was given to attributes defining the longest sizes of file names and paths on a distribution, but these were not included since their use could neither ensure failure nor success of installing or copying a particular product from the distribution. Another issue implementors should consider is the maximum name and path that may be contained within a supported archive.

The need for the *media_sequence_number* attribute is to number the tapes (or disks or whatever) if a distribution is on more than one of them. If there is only one, then its number is 1.

The following attributes at one point were listed as distribution attributes. However, it was determined that the only time it could be guaranteed that these attributes were accurate was for an initial distribution definition. As soon as a *swcopy* or *swremove* operation occurred on a distribution, the attributes could be invalid because it would be impossible to modify these attributes in any logical manner based on the operation. It is recognized that these attributes are valuable and many vendors may choose to put them in as vendor extensions.

tag
A short name associated with the distribution, used for selecting the distribution from the command line

title
A longer name used for display purposes.

description
A more detailed description of the contents of the distribution.

revision
A revision associated with the distribution.

media_type
Describes the type of media being used (for example, CD-ROM, 8 mm, etc.)

copyright
The copyright notice for the distribution.

create_time
The date, in seconds since the Epoch, when the distribution was made.

number
The vendor part number for the distribution.

architecture
A sequence of characters used by a vendor to describe the machine or product. This is presumably more ‘‘user friendly’’ than the values returned by the *uname* utility.

Usually distributions will be created upon creation of the first product with *swpackage* or *swcopy*. Usually distributions will be removed as a part of removing the last product with *swremove*. An implementation may choose to provide more explicit control for creation and deletion of empty distributions. The *swcopy* and *swremove* utilities should be used for this purpose. The *swmodify* utility may also be used.

B.2.4 Media

This section is inserted to maintain parallel numbering with the main section numbering in Chapter 2 on page 5. No additional rationale is required under this heading.

B.2.5 Installed_Software

The installed_software catalog may be located by something as simple as a pathname where the catalog is stored as a file, or it could be located in a more complicated fashion such as with a key from a directory service used to identify all or part of a database.

B.2.6 Vendor

The *vendor.tag* attribute is intended to distinguish software objects from different vendors that happen to have the same *product.tag*. A vendor should attempt to choose a *vendor.tag* that is unique among all vendors.

B.2.7 Category

This section is inserted to maintain parallel numbering with the main section numbering in Chapter 2 on page 5. No additional rationale is required under this heading.

B.2.8 Software

This class definition exists for convenience in defining the classes that inherit from it. It is not intended that any direct instances of this class be created, but only of the classes that inherit from it.

This standard has defined four related software objects:

- Products
- Filesets
- Bundles
- Subproducts

See Figure B-4 on page 174. Implementations are encouraged to present these to the user as hierarchy of similar ''software'' objects, and to actually implement these so that they differ only as needed. That is to say, an implementation should use inheritance from a common class as much as possible. The rationale for the four differently named software objects is as follows:

- Products and filesets are concepts firmly entrenched in existing practice. All of the many practices that have contributed to this standard have included these two levels. Manageable software objects necessarily includes some files to manage. This is the basis of a software product. Additionally, most application software has both required and optional pieces, so often only a subset of the product may be installed. Thus, a fileset is chosen as a ''set of files'' and a product is a collection of filesets that have a number of shared attributes, and are distributed in a single distribution (usually from a single vendor).
- It was agreed that a ''recursive notational convenience'' was very desirable. Additionally, many (but not all) existing practices had realized the need for various overlapping groupings of software into new ''configurations.'' Bundles and subproducts are merely ''macro'' or ''recursive'' products and filesets, respectively. Just as products and filesets are a bit different, the use of bundles and subproducts are a bit different. Bundles provide a way to make products out of existing products or parts of products. Subproducts provide a way to provide selectable units that may overlap in fileset contents. For example, a fileset may be part of ''runtime'' support as well as ''development'' environment subproducts. Finally, bundles and subproducts are recursive in that they may contain other bundles and subproducts, respectively.
- The containment of filesets and subproducts within products allows for derived naming of components of a product that is, a simple *tag* for a component relative to a more complex name (*tag*, *revision*, *vendor_tag*, *architecture*) for a product. In addition, this leads to distributions with a simple directory structure for filesets within products.

The need to localize the following descriptive software and vendor attributes was recognized *title*, *description* and *copyright*. However, since the existing practice for localization of software information files in portable media is immature, this has been deferred to a possible future revision of this Software Administration specification.

Until a future revision of this Software Administration specification addresses localization, one recommended way to internationalize these attributes is to create vendor-defined attributes with the format:

```
keyword.<LANG>
```

where keyword is ‘‘description.’’, ‘‘title'', or ‘‘copyright’’, and <LANG> is the value of the **LANG** environment variable. An implementation should then recognize if **LANG** is set to a value other than its default and search for a corresponding attribute. If that attribute does not exist, then the default one will be used. For example:

```
product
    tag GreatProduct
    title "This is great!"
    title.FRENCH "C’est magnifique!"
    title.GERMAN "Sehr gut!"
    description"Long boring paragraph why this is great"
    description.FRENCH "...
    description.GERMAN "...
    . . .
```

Note that the *tag*, *revision*, and other attributes that affect the defined behavior of the implementation, shall not be internationalized. For this revision of this Software Administration specification, this includes all defined attributes except *title*, *description*, and *copyright*.

The size for the software may be larger than that supported by the POSIX.1 *size_t* structure since software can contain many files. It is recommended that an implementation allocate at least 64 b for the internal storage of the software *size* attribute.

B.2.9 Products

The value of the *revision* attribute is interpreted as a . (period) separated string, as defined in Section 2.9.0 on page 17, and further in Section 3.4.1 on page 38. This definition permits the use of such a string, but does not require it. The string can be constructed entirely without the use of periods. An example of the comparison is:

```
A1.003.01 < A.004.00 < B.000.00
A1_003_01 < A_004_00 < B_000_00
First < Second < Third
First < Fourth < Second
```

Historically, some implementations computed the value of *instance_id* sequentially, while other implementations have used an algorithm based on the product *tag*, *vendor_tag*, and the various machine type attributes. No implementation is specified, other than to guarantee that the *tag* and *instance_id* uniquely identify the product within the distribution or installed_software object. This is to make it easier to specify a particular product when there are other products sharing the same *tag* as would be the case when there are different product instances in a distribution for several machine types or multiple concurrent versions on a host.

The *vendor_tag* attribute is intended to be universally unique to distinguish product and bundle software objects that otherwise would be treated as the same object if the *tag*, *revision*, and *architecture* attributes were the same. Guaranteeing universal uniqueness is difficult at best, and no need was seen at present to cause the value of *vendor_tag* to be either some sort of machine-generated universally unique value or officially registered.

Multiple versions of the ‘‘same’’ product or bundle (ones with the same value for the *tag* attribute) is supported by each version possessing values of the version distinguishing attributes

unique within that installed software catalog.

The *architecture* attribute should include information related to four *uname*( ) structure members. The *architecture* attribute is needed for `software_spec`s since the patterns used for determining compatibility in the attributes related to *uname*( ) can be somewhat complex and contain patterns, while `software_spec`s themselves can contain patterns.

It is recommended that a set of guidelines be used for the architecture attributes to maintain a consistent ''syntax'' for related architectures. This increases the usability of this field for users selecting software. An example guideline is to order any information contained in the value of the attribute in a consistent way, separated by a consistent delimiter. For example:

```
architecture sunos_4.1_sun4
```

for a product with the attributes:

```
os_name sunos
os_rev 4.1.*
os_ver *
machine_type sun4*
```

Another example is:

```
architecture hp-ux_9_pa-risc
```

for a product with the attributes:

```
os_name hp-ux
os_rev 9.*|10.*
os_ver [a..e]
machine_type 9000/[6..8]???
```

Product machine attributes describe the target systems on which this product may be installed. Each of these keywords are related to a POSIX.1 *uname*( ) member and may be defined as a simple string, or a software pattern matching notation. How compatible software is determined depends on whether the products are being installed on the system that will be using them, or whether the installation will be used by other systems with perhaps different attributes.

If a uname attribute is undefined, the behavior is essentially the same as if it were defined to be `*` (meaning compatible with all systems).

The product directory for an application should be the directory that is part of all paths in the product. Thus, if an application has three filesets that contain files below `/appl/console` , `/appl/agent` , and `/appl/data` respectively, the *product.directory* attribute should be set to `/appl` . If a user relocates the product with a command like:

```
swinstall appl,r=1.0,l=/disk2/appl
```

then all three filesets have the same location attribute. If the user relocates the product to three different locations:

```
swinstall appl.console,r=1.0,l=/disk1/appl
swinstall appl.agent,r=1.0,l=/disk2/appl
swinstall appl.data,r=1.0,l=/disk3/appl
```

then each fileset will have a different location attribute. There will be three product instances containing the three filesets (since products versions are distinguished by location), but the user can still identify all three filesets as one with the specification:

```
swverify appl,r=1.0,l=*
```

Alternatively, the user could relate all these locations with the same version qualifier, such as `"q=current"` as follows:

```
swinstall appl.console,r=1.0,l=/disk1/appl,q=current
swinstall appl.agent,r=1.0,l=/disk2/appl,q=current
swinstall appl.data,r=1.0,l=/disk3/appl,q=current
```

and subsequently identify all pieces with:

```
swverify appl,q=current
```

The *postkernel* attribute supports the ability to install one operating system in proxy (to an alternate root) by another implementation that does not understand that operating system. All products that contain kernel filesets that will be installed into the same installed_software object should have the same path defined. There should be one core OS kernel fileset that includes this path in its set of files so that it has been installed by the time the *postkernel* script is executed.

In general, a product with no `preinstall` or `postinstall` scripts is recoverable. However, if there are `preinstall` or `postinstall` scripts, then `unpreinstall` and `unpostinstall` scripts shall be provided if any steps need to be undone to support autorecovery.

There was an issue whether dependencies should be an attribute of a product. The following types of dependencies have been discussed:

- Fileset to fileset within a product.
- Fileset to (some other) product.
- Fileset in one product to fileset in another product.
- Product to product.
- Product to fileset in some other product.
- Product to fileset in that product (essentially mandatory fileset).

The last three dependency types are not necessary if the first three types exist (which they do), since those dependencies can be specified in terms of the others. For example, if an entire product depends on a second product, then the second product can be defined as a dependency for all filesets in the first product.

The developers of this Software Administration specification recognized that numerous additional dependency requirements are possible, particularly for software updates. These may be handled via `checkinstall` scripts, and can be considered for future revisions of this Software Administration specification.

The intention behind the inclusion of the *layout_version* attribute within a product is that it be required if its value is different than that for its associated software_collection .

B.2.10 Bundles

Bundles serve two purposes they allow the software supplier to group different subsets of products into new configurations or products, and they allow the software administrator to build useful groups of software (configurations) from already defined bundles and products.

The bundle class does not have *location* or *directory* attributes. This is because `software_spec`s within the bundles can refer to products with different default directory attributes or even products that have been relocated.

Bundles have ‘‘uname’’ attributes that only have any value if the bundle aggregate has a different compatibility than that of any of its contents. Besides offering more control to the person defining the bundle, it is useful in a GUI that wants to only display compatible software by default. For example, a bundle may contain one product that operates on a system with an uname attribute of ‘‘A’’ and another product that operates on systems with uname attributes of ‘‘A’’ or ‘‘B’’. In this case, it might be useful to define the bundle attribute to be ‘‘A’’. Since it is possible that not all the bundles contents exist in a particular distribution or installed_software object, it may not be possible to determine the compatibility of the bundle in all cases unless the bundle attributes are also defined.

The *vendor_tag* attribute is intended to be universally unique to prevent naming clashes for similarly named products and bundles from different vendors. Guaranteeing universal uniqueness is difficult at best; it was deemed unnecessary at present to cause the value of *vendor_tag* to be either some sort of machine-generated universally unique value or officially registered.

The intention behind the inclusion of the *layout_version* attribute within a bundle is that it be required if its value is different than that for its associated software_collection .

The value of the *bundle*contents attribute is not modified when a location is specified for a bundle, allowing future resolutions of its contents to remain consistent. For example, assume bundles `"CAT"` and `"DOG"`, and products `"FOO"` and `"BAR"`, all with directory attributes defined as `"/"`:

```
bundle
    tag CAT
    contents DOG,l=/dog BAR,l=/bar
bundle
    tag DOG
    contents FOO,l=/foo
```

When the bundle `"CAT"` is installed and relocated to `/cat`, the following objects are installed:

```
CAT,l=/cat
DOG,l=/cat/dog
FOO,l=/cat/dog/foo
BAR,l=/cat/bar
```

So, when resolving `"CAT,l=/cat"` in installed software, applying the proper locations to the `software_spec`s in the contents will result in the same `software_spec`s in the installed software.

Bundle definitions are only copied or installed when explicitly specified since they are external to the product and not always applicable to the use of the product installed. The creator of a product has no control over what bundles reference it. For example, a product may be a member of numerous bundles, and many of those bundles will likely have nothing to do with the bundles and products chosen to be installed. Also, see Section B.2.12 on page 183.

Bundles and subproducts have lists defining their contents that are always copied (*contents* is a static attribute). So, if a partial bundle or product is copied, the value of the *contents* attribute does not change. However, by comparing that attribute to what objects are actually installed, ‘‘completeness’’ of a bundle or subproduct can be determined.

B.2.11 Filesets

The *media_sequence_number* is used for serial distributions to describe which media the archive containing the fileset starts on. There is generally one archive per media, unless a fileset is larger than a media. Each media has a unique sequence number whether it begins an archive or continues a previous one.

At one point a fileset *class* attribute existed that could contain the value of recommended, mandatory, or optional. The attribute was removed because it was felt that this Software Administration specification could not specify any behavior for the attribute. It would be possible to make a specific fileset mandatory by having all other filesets in that product specify it as either a prerequisite or corequisite.

Another way to handle recommended, mandatory, or optional filesets would be to create subproducts with *tag*s of the appropriate names. Although this Software Administration specification does not specify any behavior based on the name of *subproduct* tag, a specific implementation could define behavior as an extension.

When there is a dependency on a software item that is the ancestor of a new software item, it is desirable for the standard to allow that new software item to meet that dependency. This is more complex than might initially be apparent, since dependencies and ancestor definitions involve ranges of revisions and other expressions. The working group felt a separate ''supersedes'' attribute might be a better solution, so explicit supersede control is separate from the ancestor/match_target functionality. Requiring supersedes to be fully qualified software specs only would help eliminate the ''pattern-to-pattern'' comparisons.

Patches do not have to supersede other patches in the same fileset. This allows ''point'' patching: separate patches that patch separate parts of the same fileset.

Patches that supersede all previous patches (cumulative patches) can be specified with a single software_spec if a patch strategy uses the same product and fileset tags, and an ascending revision numbering scheme.

Example:

```
supersedes   product.fileset,r<revision
```

B.2.12 Subproducts

Unlike bundles, subproduct definitions (that are internal to a product) are copied or installed when any fileset specified in the *contents* attribute of the subproduct is copied or installed. Products are meant to be sets of related software and are usually created and managed by one person or organization. Additionally, subproducts are normally used to specify useful subsets of filesets within the product, which in turn are useful for dependencies. With subproducts, the ''parts make up the whole.''

B.2.13 Software_Files

This class definition exists for convenience in defining the classes that inherit from it. It is not intended that any direct instances of this class be created, but only of the classes that inherit from it.

The *compression_type* attribute allows compressing and uncompressing of individual files during *swcopy*, and uncompressing during *swinstall*. The way in which an implementation uses this attribute is undefined, although the general thought was that this would normally be the name of a compression/uncompression routine with a simple interface.

An implementation should be flexible in locating routines specified by compression_type, utilizing any or all of the following:

- Built-in knowledge of the *compression_type* format for compressing and uncompressing
- The product control directory for a program named in *compression_type*
- **PATH** on the target system for a program named in *compression_type*

No particular compression method is specified in the standard largely because the developers of this Software Administration specification saw no standard for file compression and did not want to specify all of the details of the compression methodology as part of this Software Administration specification. It was generally agreed that to achieve adequate interoperability, a single method of consensus should be supported by all implementations. It is likely that the format used by the *gzip* utility is appropriate for all implementations. Each implementation may support any number of other methods.

The interface to the compression routine was also left unspecified. It is recommended that input be taken from stdin and output be directed to stdout, that the routine operate with no option to imply compress, and that a `-u` option imply uncompress. However, specific compression routines may require more complex interfaces.

The group also considered archiving of compressed files, that is, concatenation or other combination into a single file. The main purpose of this would be to save cluster space on diskette distributions. It was finally decided that the risks for current standardization were too high especially if an archive extended over more than one diskette and the issue was left implementation dependent. In implementing this, there should be consideration of the following factors:

- A new *archive_source* attribute to indicate that the file contents are within a named archive.
- Defining a new fileset *archive_type* attribute with values of empty string, `cat`, or the name of an archive routine like *tar*. The type `cat` indicates simple appending to an archive file. If `cat` (or even possibly `tar`) were used, an *archive_offset* attribute would indicate where within the archive the file started. This could be used for fast single-file extraction using either *size* or *compressed_size*.
- Extended options on *swcopy* for *archive_files* and *archive_type* (similar to *compress_files* and *compression_type*). The *uncompress_files*true option on *swcopy* would both unarchive and uncompress.
- An archiver interface that permitted appending or extracting one file at a time.
- The archiver, like the compressor, could be distributed in the product control directory.

Finally, compression support for *swpackage* was considered, and deemed as unnecessary, since compression can be achieved by copying after packaging. But an implementation can easily add attributes to achieve this function.

B.2.14 Files

The letters chosen for the file *type* attribute are consistent with the syntax of the *find* utility with the `-type` option, as defined in POSIX.2. Hard links are not specifically mentioned in POSIX.1 section 5.6.1.1. Symbolic links are not mentioned in POSIX.1 but are included to support existing practice. Work to standardize symbolic links is included in referenced document [B21]".

Implementations running on operating systems that do not support a POSIX.1 file system can interpret the defined attributes in any appropriate way. Any implementation can extend file attributes with additional attributes appropriate to the file system in question. To avoid confusion when defining new attributes for a particular file system, it might be best to prefix such attributes with a designator of the file system. An example, for a FAT file system, might be the attributes *FAT_Hidden* and *FAT_Readonly*.

There was some debate whether the *major* and *minor* attributes are appropriate or not since there is no standard that specifies how these files are created. In addition, this Software Administration specification specifies that the serial distribution be in POSIX.1 *cpio* or *tar* format; however these attributes are biased towards *tar* format as opposed to *cpio* format.

Considered was a size file type (`z`) that was removed in favor of the `space` control_file similar to SVR4. An implementation may choose to internally implement a size type or a separate size_file object to represent the data from this file.

The developers of this Software Administration specification considered an *is_exclusive* (directory) attribute that was removed due to objections that the utilities would remove files that they did not have recorded in their database. Also, this was not a common need, and can be implemented either as vendor extension or by having software `fix` script implement similar functionality.

There has been much discussion about compression being handled within the scope of this Software Administration specification. Currently there are ways that both implementations and individual software products can handle compression. Compression can be handled through cooperation of the source and target roles, if they are from the same implementation. Software vendors can choose to ship their files compressed and uncompress them as part of the `postinstall` script. They can add a `space` control_file to account for the extra space required.

A similar need would apply to other post processing, such as for ANDF files that are processed as part of postinstall or configuration.

Though it may be adequate for protecting against accidental damage, the existing POSIX.2 *cksum* is considered inadequate for virus protection. Implementations may wish to create additional vendor-defined attributes and utility behaviors for this purpose.

Each of the prerequisite or corequisite `dependency_spec`s in the list is required to resolve successfully in order for dependencies to be met. Also, a `dependency_spec` can contain alternate `software_spec`s separated by the `|` (vertical line) character (see Section 3.4.1 on page 38). So, if a fileset has a corequisite dependency on software, expressed with a Boolean equation `(A|B|C)&(D|E)`, this can be specified in a PSF as:

```
corequisite A|B|C
corequisite D|E
```

There are files (particularly for OS software such as `/etc/rc` for SVR4 and `autoexec.bat` for DOS) that are modified between software update times. These may be termed modifiable files. Although OS modifiable files are slowly being replaced by mechanisms where applications can simply add their own requirements as separate read-only files in a particular directory, there currently would be some value in supporting features where modifiable files are compared with

the original files to see what changes need to be applied during software updates. Actually implementing these changes is a more difficult problem since it requires knowledge of the formats of the files being updated. Similarly, reversing (during *swremove*) changes made to modifiable files (during *swinstall* and *swconfig*) is an exceedingly difficult problem. The existing practice for treating modifiable files is fairly ad-hoc. It was not feasible to address all of the possible needs for updating modifiable files. Instead, it does provide the attribute *is_volatile* for files that may be modified after installation, and leaves the rest of the treatment of modifiable files as either implementation defined, or handled in control scripts. This area may be considered for a future revision of this Software Administration specification.

B.2.15 Control Files

Using *tag*s as the identifier of when a script should be executed (independent of the path the script is stored as) allows anywhere from one file per *tag* to one file for all *tag*s. A concern on PC or DOS systems is that requiring more than one control script for all *tag*s is a space problem. Instead, software vendors might prefer a single master script that took care of all needs. Multiple scripts are also supported, since many software vendors favor this approach over a ‘‘mega-script.’’ However, other vendors may prefer the single script approach, especially to save space if there are many scripts defined for this product that share a lot of the same code.

Control files do not have *mode*, *owner*, *group*, *uid*, *gid*, and *mtime* attributes since they are not necessary for the execution of the control scripts or for the management of these files within the distribution or installed software catalog. However, an implementation shall ensure that they are executable.

The *interpreter* attribute has two uses. It is useful for those who choose not to use the POSIX.2 shell, that is, *sh*. It is also useful for systems that would not otherwise require POSIX.2. Those creating distributions and control files are encouraged to use the POSIX.2 shell for portability.

B.3 Common Definitions for Software Administration Utilities

B.3.1 Synopsis

No additional rationale is required under this heading.

B.3.2 Description

No additional rationale is required under this heading.

B.3.3 Options

The `-d` option is needed to remove ambiguity for utilities that operate on both distributions and installed software.

The `-r` option is needed for the following reasons. Installing software at `/` involves a somewhat different set of operations than software installed at an alternate root, as well as a different implied use. Software installed on alternate roots is not configured in the context of the target where the software is installed, but rather in the context of the client actually running the software. Another difference is that the target is not rebooted after installing software that requires a reboot (the clients of the software need to be rebooted). An alternate root containing operating system software can be thought of as a root to which one could *chroot*(). From a usability standpoint, it is important that alternate roots are understood to be different than relocating a software product, or specifying an alternate catalog for the same root.

Related to the `-s` option, an implementation could define an additional source syntax to use well-known sources whose existence is available through some sort of directory service.

The `-s` option could be extended to supported multiple source specifications. There are several possible ways to interpret multiple sources, including searching sources sequentially, ignoring all specifications after the first one, using the last specification, or choosing the ‘‘best’’ source based on criteria such as performance or ability to reduce network load. It may even be desirable for multiple source specifications to be interpreted differently for different commands.

An implementation may implement the `-p` option (preview) by simply executing the command through the analysis phase. Alternatively, an implementation may emulate the execution phase, listing the operations that would occur, including listing control scripts that would be run, but not actually performing those operations. As preview is undefined, other alternatives are possible.

B.3.3.1 Non-Interactive Operation

It is recognized that there may need to be some sort of interaction with the user in order to handle multiple volumes (for example,for example, tapes) for sources and targets.

B.3.4 Operands

The 1387.2 developers concluded that the `@` character does not have any applicable precedence as a separator of operands, so the use of `@` in mail addresses and BSD commands is a bit different. Another point was that having two lists of operands was not desirable in any case.

On the other hand, the two types of operands are the two key objects upon which the utilities operate. The syntax is valid according to the utility guidelines from POSIX.2 section 2.10 2. Distributed utilities extend the problem space that POSIX.2 has already addressed, thus the need for precedence might be less. Thus, it was decided that the `@` was acceptable, and perhaps desirable over the alternatives.

One alternative was to move one or both operands to options (such as `-S` for software and `-T` for targets). But, it was felt that this was not necessary because there are already `-f` and `-t` options for files containing lists of operands. Another point was that listing target operands on the command line was not critical in any case, as an administrator of many systems would not use either the *@ targets* or `-T` *target* `syntax`.

B.3.4.1 *Software Specification and Logic*

Using a less formal grammar convention that defines zero or one item by enclosing these items in `[]` (brackets) and zero or more repeated items in `{}` (braces), the following shows a common subset of the `software_spec` syntax:

```
software_spec   : bundle_tags [ product_tags ] [ version ]
                | product_tags [ version ]
                | '*' [ version ]
                ;
bundle_tags     : bundle { '.' bundle }
                ;
product_tags    : product
                  [ '.' subproduct { '.' subproduct } ]
                  [ '.' fileset ]
                ;
version         : { ',r' rel_op revision }
                  [ ',a=' architecture ]
                  [ ',v=' vendor_tag ]
                  [ ',l=' location ]
                  [ ',q=' qualifier ]
                | ',*'
                ;
rel_op          : '==' | '!=' | '>=' | '<=' | '<' | '>'
                ;
```

The keywords **bundle**, **product**, **subproduct**, and **fileset** refer to the *tag* attributes of those objects. The value of `revision` is usually a dot separated string compared to the value of the *revision* attribute of the first object. The values of `architecture`, `vendor_tag`, `location`, and `qualifier` are usually exact strings or patterns compared to the like-named attributes of the first object. These version attributes can validly be specified like `revision` is, but operators and multiple specifications do not make much sense.

Examples of `software_spec`s are:

```
*
Networks
Networks.X11
Networks.X11.Runtime,a=*80?86*
X11
X11.Runtime
X11.Runtime,r=4,v=CloneInc
X11.Runtime,r>=4.0,r<5.0
X11,r=4.03.07,l=/usr/X11R4
X11,r=5.00,l=/usr/X11R5,q=latest
X11,*
*,a=*80?86*
```

A `software_spec` shall begin with a bundle or product *tag*. A particular bundle or product object can be determined since they share the same name space (they also have different *instance_id* attributes).

The *location* attribute applied to the product means all filesets in that product in that location. This is the same set of filesets as if the *location* attribute was applied to the filesets.

Since the components of the `version_qualifier` of a `bundle_software_spec` refer to the attributes of bundle objects, there is no way to select one version of a product if more than one version is specified in the *bundle*contents. Neither the inclusion of multiple versions of a product within a bundle, nor the specifying of partial bundles, is seen as the normal use model, so having this Software Administration specification limit the flexibility slightly in this area was deemed as acceptable.

This Software Administration specification permits the use of values other than those defined in Section 3.4.1 on page 38 for `ver_id`. This Software Administration specification also permits the use of `ver_id` in conjunction with attributes and objects other than the first listed in a `software_spec`. This allows additional flexibility for identifying software objects.

A possible syntax for these vendor extensions include, but are not limited to:

ver_id	attribute	object
br	*revision*	bundle
ba	*architecture*	bundle
bv	*vendor_tag*	bundle
bl	*location*	bundle
pr	*revision*	product
pa	*architecture*	product
pv	*vendor_tag*	product
pl	*location*	product
fr	*revision*	fileset
fl	*location*	fileset

The `ver_id fr` is seen as most useful since it can identify a particular fileset object within a product where the product may not have a revision, but the fileset does. Note however, that any object can still be identified with only the attributes defined in this Software Administration specification. For example, if a bundle includes two partial products with the same *tag* value but different revisions or locations, these partial products could be identified with the standard syntax by excluding the bundle portion of the `software_spec`. For example, `bundle.product,pr=1.3` could also be identified by a `software_spec` of `product,r=1.3`.

In another example, the fileset that could be identified by its revision (`product.fileset,fr=1.3`) could also be identified by a `software_spec` (for example, `product,r=revision`), where revision refers to the product revision, including possibly the empty string.

Relocation occurs by replacing the *product.directory* part of each file path as it occurs in the distribution, with the *location* specified and using the resulting path for installation. This is still relative to the installed_software directory described below. See *swinstall* on page 96 for more information.

Using a `sw_pattern` in a `software_spec` is a way for the user to indicate that all software objects that match the `software_spec` are to be included. For example, applying the

software_spec ``*'' to *swcopy* means to copy all software in the distribution. Applying the software_spec ``Foo,*'' to *swremove* means to remove all versions of Foo.

The behavior for *swlist* is different (by default including all software if none is specified) because this is the command that is used to find all versions of software, and because listing cannot negatively affect the state of the software_collection.

If using software pattern matching notation characters on the command line, they shall be escaped or enclosed in single quotes to avoid matching files in the current working directory.

This specification provides the means to select products and specify dependencies using a single syntax. The use of the shell-type pattern match specified in POSIX.2 section 3.13 allows for reasonable specification of sets of values that share such patterns. Thus, for example, a specification of "a=HP-UX*" may be used to select packages for any of a set of architectures. The specification using the relational operators provides support for testing the type of release/version specifications that are frequently used by vendors. In particular, it provides support for testing when a numeric test is needed (for example, comparing 2.9 to 2.10 as version levels of a product). Additional operators such as >> were considered. The specification of the >> operator allows the user to specify the selection of the most recent (highest version number) of a set of otherwise identical packages. This exposes to the interface the mechanism used by *swinstall* to select such a package.

The range of attributes that may be specified allows for selection of packages that may be needed to support code serving to alternate architectures, or other operating environments. In addition, it provides the needed support to specify installed software that may only be distinguished by the location of installation.

Examples of fully-qualified software_specs are:

```
Foo,r=3.0,a=,v=XT
BundleA.Foo,r=1.0,a=,v=XT
Dow.Bar,r=2.0,a=SunOS,v=,l=/opt/foo.2
```

It is possible for bundles to contain software_specs that are not fully-qualified. This is not recommended for bundle definitions provided by software vendors because the results of operations on this bundle may be undesirable for an administrator. However, there is some flexibility provided by ambiguous software specs that administrators may want to use.

For example, a bundle with contents ``*.Man'' could be used to manipulate all ``Man'' filesets or subproducts in all products.

If a vendor includes any wildcards in a software_spec in a bundle definition, then the *vendor_tag* attribute should be included and its value should have no wildcards, thus limiting the scope of the pattern matching.

The difference between ``FOO,v='' and ``FOO'' is that the first will only match a product or bundle ``FOO'' where vendor is not defined, while the second will match a product or bundle ``FOO'' with any vendor definition.

B.3.4.2 Source and Target Specification and Logic

Using a less formal grammar convention that defines zero or one item by enclosing these items in [ ] (brackets) and zero or more repeated items in { } (braces), the following shows a common subset of the software_collection_spec syntax:

```
software_collection_spec    : [ host ] [ ':' ] [ path ]
```

Examples of distribution software_collection_specs are:

```
/var/spool/sw
hostA
hostA.cloneinc.com
hostA:/var/spool/sw
15.1.94.296
15.1.94.296:/depots/applications
```

Examples of installed_software `software_collection_spec`s are:

```
/
hostA
hostA.cloneinc.com
hostA:/
15.1.94.296
15.1.94.296:/exports/applications
```

Target distributions in the serial format need not be supported for *swverify*, *swremove*, and *swmodify* as this requires the implementation to unload the entire distribution, merge in the changes, then reload it. The user can accomplish this (and an implementation can implement this) by first copying the distribution into a directory format, implementing the changes, then copying the distribution back to the serial media. This operation also could require significant temporary disk space.

A similar rationale applies to *swcopy*, and *swpackage*, which by default, overwrite the existing distribution instead of merging in the specified software.

B.3.5 External Influences

B.3.5.1 Defaults and Options Files

For SVR4 or similar file system layout, the defaults file may be located in `/var/adm/sw/defaults`. The use of this location is strongly encouraged.

The 1387.2 developers considered the difference between ''system-level'' defaults and ''site-level'' defaults.

The former is provided by the implementation of the utilities and the latter is constructed by the administrator. The intent here is for the implementation to respect any customizations to the system level defaults file, so it can be used for site policies. It is recommended that implementations ''hard code'' the defaults as opposed to relying on the system file containing all definitions, and provide a means to support new options in future releases without changing the site specific values in the system defaults file.

B.3.5.2 Extended Options

For SVR4 or similar file system layout, *distribution_source_directory* may be set to `/var/spool/sw`. The use of this location is encouraged.

For SVR4 or similar file system layout, *distribution_target_directory* may be set to `/var/spool/sw`. The use of this location is encouraged.

For SVR4 or similar file system layout, *installed_software_catalog* may be set to `/var/adm/sw/catalog`. The use of this location is encouraged. The catalog may simply be a pathname of a directory where the database containing the catalog is stored, or may be a key into a directory service specifying a catalog in a file or database, or any other implementation-defined method of specifying a catalog. The location of the storage for the catalog itself is implementation defined.

The *swinstall* utility, and other utilities that operate on installed_software, modify the catalog information based on the outcome of the utility. Information contained within the catalog is resolved in the context of each target.

Originally, it was thought that a catalog would be kept as a flat file in a directory that could be specified using this option. In the interest of generality, so that implementors might be allowed to use databases, the *catalog* attribute is now described as a key. This allows an implementor to either use a flat file or a database or some other form of persistent storage for the information, yet still be able to separate the address space as desired. The motivation for permitting the separate address space stems from the following two cases. First, it seems desirable to allow ordinary (non-root) users to be able to use *swinstall* to store software in their own private space. Likely the only real restriction is a potential lack of write authority to the central storage for the catalog, hence the ability to create a separate catalog. This also allows a user to manage personal software with utilities such as *swremove* or other utilities. Second, installations may wish to deploy stable versions of their software in the normal location, and a test version installed in a second location where access may be more tightly controlled. There may even be other versions installed that are under development. Since this software may have identical attributes, it is desirable to allow such separate space for management. Both of these examples show the need for separate domains of software management.

Two values of *autoselect_dependencies* (*autoselect_dependencies=true* and *autoselect_dependencies=as_needed*) support different possible policies by the user. Having *autoselect_dependencies=true* ensures that all targets are kept in sync, while having *autoselect_dependencies=as_needed* prevents the possibility of updating dependency software to a higher revision unnecessarily.

Autoselection of a dependency across products is possible if a compatible product version with the highest revision that meets the dependency is unique. In other words, the same rules apply for dependency selection as for normal selection as described in Section 3.4.1 on page 38.

For the *ask* option, the `checkinstall` and `configure` scripts are required to detect needed `response` files when they are necessary, and return with the appropriate warning or error.

For the *installed_software_catalog* option, the catalog and the directory together form a key to identify one installed_software object. For example, this would allow the files on the file system to be split up into different management domains. For example, OS software, networking software, and application software could be in three different logical installed_software objects, although they are all installed under the root file system.

B.3.5.3 Extended Options Syntax

In the interest of having a single common extended option syntax for all the POSIX system administration standards, the following syntax was agreed upon. As of this writing, the syntax is a superset of that used by this Software Administration specification, IEEE P1387.3, and IEEE P1387.4.

```
%token          FILENAME_CHARACTER_STRING  /* as defined in 2.2.2.37  */
%token          NEWLINE_STRING             /* as defined in 2.2.2.61  */
%token          PORTABLE_CHARACTER_STRING  /* as defined in 2.2.2.68  */
%token          SHELL_TOKEN_STRING         /* as defined in 2.2.2.80  */
%token          WHITE_SPACE_STRING         /* as defined in 2.2.2.110 */

%start  sysadmin_option
%%

sysadmin_option          : qualifier option operator_value
                         ;
```

```
qualifier                : compulsory_qualifier  command_qualifier
                         ;

compulsory_qualifier     : /* empty */
                         | '-' | '='
                         ;

command_qualifier        : /* empty */
                         | command '.'
                         ;

option                   : keyword op_ws
                         ;

operator_value           : '=='
                         | value_qualifier '=' value
                         ;

value_qualifier          : /* empty */
                         | '+' | '-'
                         ;

value                    : op_ws value ws single_value
                         | op_ws single_value
                         ;

single_value             : value_structure
                         | SHELL_TOKEN_STRING
                         ;

value_structure          : '{'  op_ws value_list op_ws '}'
                         ;

value_list               : /* empty */
                         | value_list ws single_value
                         | single_value
                         ;

command                  : FILENAME_CHARACTER_STRING
                         ;

keyword                  : SHELL_TOKEN_STRING
                         ;

op_ws                    : /* empty */
                         | ws
                         ;

ws                       : WHITE_SPACE_STRING
                         ;

%start  command_line_options
%%
command_line_options     : command_line_options  ws sysadmin_option
                         | sysadmin_option
                         ;

%start  options_file
%%
```

```
options_file            : options_file NEWLINE_STRING option_file_line
                        | option_file_line
                        ;

option_file_line        : op_ws op_comment
                        | op_ws sysadmin_option op_ws op_comment
                        ;

op_comment              : /* empty */
                        | '#' PORTABLE_CHARACTER_STRING
                        ;
```

Notes:

1. A `-` (hyphen) qualifier indicates a compulsory behavior while `=` (equal) indicates a non-compulsory behavior.
2. For options that support multiple values, values can be added to the existing list of values by using the `+=` (plus equal) operator. Similarly, values can be removed by using the `-=` (hyphen equal) operator. Any option can be set to the default value by using the `==` (equal equal) operator and value combination.
3. A shell token can be an unquoted or quoted string according to the rules of token recognition rules described in POSIX.2 section 3.3 (token recognition). For example, it can use single or double quotes and can contain like quotes if escaped with backslash. It can also support the same level of internationalization as the POSIX shell.
4. The multiple value convention is consistent with white space separating tokens in commands (operands) and allows commas to be used in the `single_value`. This also allows multiple values to be specified without using quotes (although quotes are still needed for multiple values on the command line).
5. If the extended option specification contains any white space at all, then the entire specification shall be quoted if used on a command line. This is because the `-x` option, which conforms to POSIX.2, requires exactly one value that is then processed using the above syntax.
6. When specified on the command line, multiple option specifications can be included after a single `-x` option if included in quotes and separated by spaces. Multiple `-x` options may also be used.
7. For option and defaults files, blank lines and all comment text [any sequence of characters beginning with an unescaped `#` (pound) and continuing through the end of that line] are ignored according to the shell token recognition rules as described in POSIX.2 section 3.3.

Precedence for Option Specification

The first rule defines typical precedence of system defaults, then a user defined set of defaults, then per task exceptions or specifications. The second rule supports normal use models of defining multiple ''sets'' of target_selections and software_selections, and being able to operate on the union of those sets. Also, the `-f` and `-t` options are simply another form for specifying operands, and are at the same level of precedence, and are thus combined with other selections. The third rule is generally an error, and the behavior is undefined (that is, it may be an error, or an implementation may chose to implement last- or first-wins). For example on HP-UX:

```
$ cc -O -g x.c
$ cc: warning 414: Debug and Optimization are mutually exclusive.
    -g option ignored.

$ cc -g -O x.c
$ cc: warning 414: Debug and Optimization are mutually exclusive.
    -O option ignored.
```

It might be convenient to have a mechanism to allow the system administrator to define a default in the system defaults file that cannot be overridden by a user. Such a function may be supplied by an implementation as an extension. This may also be considered as part of a future revision to this Software Administration specification.

B.3.5.4 Standard Input

No additional rational is required under this heading.

B.3.5.5 Input Files

No additional rational is required under this heading.

B.3.5.6 Access and Concurrency Control

If the *installed_software_catalog* is referenced by a path on the file system, then the user can create a catalog in their own user work space (creating their own installed_software object), and install and manage software in that installed_software object.

If the catalog is stored in a file, then a corresponding ability to create an installed_software object (and thus, a catalog) is needed.

Access control includes such things as requiring particular authority to operate on particular software or software_collections. Concurrency control is the prevention of more than one writer at a time to the catalog or data areas. Restrictions to prevent multiple concurrent writers were originally part of the draft, but later determined to be excessively restrictive. It is conceivable that more than one writer could safely be active at a time if the work involves no common files. Failure to allow multiple concurrent readers of the catalog, or other data files, is strongly discouraged.

There are two aspects to access control as follows:

- Those related to file system access and hence determined by the operating system (that is, the ability to write the files described by the sofware file objects).
- Any additional access control on the software objects, including access to (possibly remote) software collections.

For access control to the files themselves, this Software Administration specification defaults to the file permissions defined by POSIX.1. File attributes are defined for the files, hence the implementation will set the POSIX.1 file permissions based on those values. Deviations from this model are permitted only for implementations running on file systems that are not POSIX.1 conformant, and then only as long as the implementation documents the resulting behavior.

Any additional access control to the software objects defined in this Software Administration specification (for example, permission to install specific software into specific software collections on specific hosts), is undefined. An implementation may choose to have no access control. For example, anyone may install any software to any system as long as the previous POSIX.1 file permissions are satisfied. An implementation may also choose to provide both authorization and authentication for access to all software objects and hosts, as well as a

distributed interface for managing the access control lists.

Like the definition of the model to implement distributed aspects of this standard, access control beyond that required by the underlying operating system is undefined. It was determined that both of these rely on technologies that have not been formally standardized, and may better be addressed in other forums.

B.3.6 External Effects

B.3.6.1 Control Script Execution and Environment

The provision for interpreters other than *sh* was requested by users among the developers of this Software Administration specification, as well as producers representing systems that might lack a POSIX.2 or even POSIX.1 operating system. By making this provision, many felt that a greater degree of acceptance and usefulness could be gained.

The restrictions placed on the option syntax are such that each of the options can be easily parsed and hence set by a control script by simply sourcing the file. The term ‘‘sourcing’’ as used here implies the use of the ‘‘.’’ command in the POSIX.2 shell. For example, the following are formats for the SW_SESSION_OPTIONS file that can be sourced:

```
loglevel=1
enforce_dependencies=false
software="A B C"

loglevel=1
enforce_dependencies=false
software="
A
B
C"
```

It is possible for the scripts to determine the *loglevel* for the command from the file pointed to by **SW_SESSION_OPTIONS**, and use that to affect the amount of stdout generated.

An implementation may have an implementation-defined user controllable behavior that invokes error handling procedures in the case of warnings returned from script execution.

The purpose of the environment variables is to pass vital information to the scripts so that they may operate appropriately under different circumstances. For example, they may want to take very different actions when **SW_ROOT_DIRECTORY** is some value other than `/`. It has also been discussed that there may need to be some way to pass other information to these scripts such as option values specified in the defaults and options file that control policy. This can be achieved with the **SW_SESSION_OPTIONS** variable, which points to a file containing all the options passed to the command, including options, selections, and targets.

One reason that this Software Administration specification differentiates install and remove scripts from `configure` scripts is to separate installing software from configuring software for actual use. This supports installing software to alternate root directories on servers for use by clients that configure that software.

The developers of this Software Administration specification also discussed, but did not include, the use of several of these variables for setting the value of specific utility options when the utilities are called from control scripts. These variables are as follows:

SW_ROOT_DIRECTORY
Could be used to specify the *directory* portion of all *target* operands.

SW_LOCATION
Could be used to specify *product*location portion of all *software_selection* operands.

SW_CATALOG
Could be used to specify the value of the *installed_software_catalog* option.

The scripts need to be aware of the environment under which they are operating. The environments that these scripts run under are as follows:

- All scripts

  Each script shall be passed its script *tag*, the root directory to which installing, the product directory where the product is located, and the control directory where the script is being executed from, as the environment variables **SW_CONTROL_TAG**, **SW_ROOT_DIRECTORY**, **SW_LOCATION**, and **SW_CONTROL_DIRECTORY**, respectively.

- `preinstall`, `postinstall`, `preremove`, `postremove`, `unpreinstall`, `unpostinstall`

  The `install` and `remove` scripts are run when loading or removing the software, or when recovering from a failed install. These may be executed by *swinstall* or *swremove* running on a host with a different architecture from the software. So, only the set of POSIX.2 utilities are guaranteed to be available on the server. Since the architecture of the file server is not necessarily known, the path to these commands is passed to the scripts via the environment variable **SW_PATH**.

  Additionally, these scripts need to know the alternate root directory so that operations are within the context of that root, not the root of the file server. (This directory is supplied to the scripts via an environment variable **SW_ROOT_DIRECTORY**.) It is critical that **SW_ROOT_DIRECTORY** is honored by these scripts.

- `checkinstall`, `checkremove`, `verify`

  It is expected, but not assumed, that these scripts mostly check the state of a system that will actually run the software. For the install check (`checkinstall`) script, again only a minimum set of commands is guaranteed to be available. This is because all check scripts are executed before any new software is installed. For the remove check and verify scripts (`checkremove` and `verify`), files from this software and its prerequisites are guaranteed to be available.

  Because of alternate root installation, these scripts also need to be aware of the **SW_ROOT_DIRECTORY**.

- `configure`, `unconfigure`

  These scripts configure the system for the software. Therefore they can run architecture specific commands, including files that are part of the software that defines the scripts. They are only run within the context of the system that will actually use the software. The **SW_ROOT_DIRECTORY** will always be `/`.

Control Script Behavior

Control scripts allow vendors to perform tasks and operations, in addition to those that the tasks perform. The *swinstall*, *swverify*, and *swremove* utilities may each execute one or more vendor-supplied scripts. The presence of these scripts in the distribution is optional. Vendors of software to be installed need only provide those scripts that meet a particular need of the software. The following summarizes the standard scripts:

request (Request script)

This is the only script that may be interactive. This script may be run by *swask*, *swinstall*, or *swconfig* after selection, and before the ‘‘analysis’’ phase in order to request information from the administrator that will be needed for the `configure` script when that script is run later.

This script is executed on the manager role and it is the responsibility of the script to write all information into the `response` file in the directory where the script is being executed (the **SW_CONTROL_DIRECTORY**). The utilities will then copy this file to the **SW_CONTROL_DIRECTORY** on the target role where the `configure` and other scripts are executed from. This `response` file may be used by any other scripts, but particularly the `configure` script.

A recommended syntax for the `response` file is a set of attribute-value pairs that can be easily sourced by the `configure` script. For example, a `request` script might look like the following:

```
echo "Enter path to locate the database"
read $path
echo db_path=$path >> $SW_CONTROL_DIR/response
```

and the using `configure` script:

```
$db_path=$default_db_path
if [ -f $SW_CONTROL_DIR/response ]
then
    . $SW_CONTROL_DIR/response
fi
create_db ( $db_path )
```

response (Response file)

The `response` file is generated by the `request` script and located in the same directory as that and the other scripts. The format and content of the `response` file is vendor defined. For example, it may be a list of environment variable definitions so that the consumer script can just source this file.

checkinstall (Install check script)

This script is run by *swinstall* during the ‘‘analysis’’ phase in an attempt to ensure that the installation (and configuration) succeeds. For example, the OS run state, running processes, or other prerequisite conditions beyond dependencies may be checked. Running this script shall be free of side-effects, for example,for example, processes may not be killed, files may not be moved or removed, etc.

preinstall (Install preload script)

This script is run by *swinstall* prior to loading the software files. For example, this script may remove obsolete files, or move an existing file aside during an update.

This script and the next script are part of the ''load'' phase of the software installation process. Within each product, all `preinstall` scripts are run (order is dictated by any prerequisites), all filesets are loaded, then all `postinstall` scripts are run.

postinstall (Install postload script)

This script is run by *swinstall* after loading the software files. For example, this script may move a default file into place.

unpostinstall (Recovery postload script)

This script is run by *swinstall* before restoring the software files if the `postinstall` script has been run. It can be used to undo the actions of the `postinstall` script.

unpreinstall (Recovery preload script)

This script is only run by *swinstall* after restoring the software files if the `preinstall` script has been run. It can be used to undo the actions of the `preinstall` script.

verify (Verify script)

This script is run by the *swverify* command any time after the software has been installed or configured. Like other scripts it is intended to verify anything that the commands do not verify by default. For example, this script may check that the software is configured properly and has a proper license to use it.

fix (Fix script)

This script is run by the *swverify* command when the `-F` option is used. Its purpose is to correct any problems reported by the `verify` script.

checkremove (Remove check script)

The remove check script is run by *swremove* during the remove ''analysis'' phase to allow any vendor-defined checks before the software is permanently removed. For example, the script may check whether anyone is currently using the software.

preremove (Preremove script)

This script is executed just before removing files. It may be destructive to the software being removed, as removal of files is the next step. It is the companion script to the install postload script (`postinstall`). For example, it may remove files that the install postload script created.

This script and the next script are part of the ''remove'' phase of the software remove process. Within each product, all remove `preremove` scripts are run (in the reverse order dictated by any prerequisites), all files are removed, then all remove `postremove` scripts are run.

postremove (Postremove script)

This script is executed just after removing files. It is the companion script to the the install preload script (`preinstall`). For example, if this was a patch fileset, then the install preload script may move the original file aside, and this remove postload script may move the original file back if the patch was removed.

configure (Configure script)

This script is executed by *swinstall* after all software has been installed (including loading files and running `postinstall` scripts) if software is being installed at `/`. It is also may run by *swconfig*, even if the software has already been configured (allowing the administrator to reconfigure software).

unconfigure (Unconfigure script)

This script is executed by *swremove* before any software is removed (including removing files and running `preremove` scripts), if removing from software installed at `/`. It is may also be run by *swconfig*.

Other scripts

The vendor may include other control scripts, such as a script that is sourced by the above scripts, or scripts not defined in this Software Administration specification. The location of the control scripts is passed to all scripts via an environment variable **SW_CONTROL_DIRECTORY**.

Control Script Return Code

This Software Administration specification only specifies return codes from scripts that affect operation of the utilities. If the script writer wants to convey additional information to the user, such information should be written to stderr or stdout, which gets recorded by the target role logging. See Section 3.6.5 on page 69.

A implementation that supports additional behaviors may initiate those behaviors based on implementation-defined return values from control scripts.

B.3.6.2 Asynchronous Events

Control script execution and file operations only generate note events at the beginning of the step, not at both the beginning and the end. This is because there is an event immediately after each script or file completes (for example, the next file beginning, or the end of fileset execution). Additionally, if there is any error with the script or file, there is a warning or error event generated after the file completes.

B.3.6.3 Stdout

No additional rational is required under this heading.

B.3.6.4 Stderr

There is a variety of warnings and errors that could occur at the target role. It is desirable that this information be communicated back to the management role and displayed on stderr of the management role. Of course the detail of the information is determined by the *verbose* level. However, since this Software Administration specification does not specify the communication mechanism between the management and target roles, it should not place unnecessary requirements on this communication mechanism, especially since this could be occurring in a distributed environment with the management role communicating with multiple target roles. Thus the only requirement here is that the target role be able to communicate a binary fail/success back to the management role.

B.3.6.5 Logging

For the management role, the information placed in the logfile is equivalent to the information sent to stderr and stdout if the *verbose* level was the same value as the *loglevel*.

The stdout, stderr, and logfile output should contain the severity of the event as part of the output message. For example:

```
NOTE:    The analysis phase succeeded on target "zook:/".
WARNING: Target "zook:/":  2 configure scripts had warnings.
```

B.3.7 Extended Description

B.3.7.1 Selection Phase

An implementation may define other valid source specifications, such as ''well-known'' sources that may be available via a directory service or an object request broker.

B.3.7.2 Analysis Phase

No additional rational is required under this heading.

B.3.7.3 Execution Phase

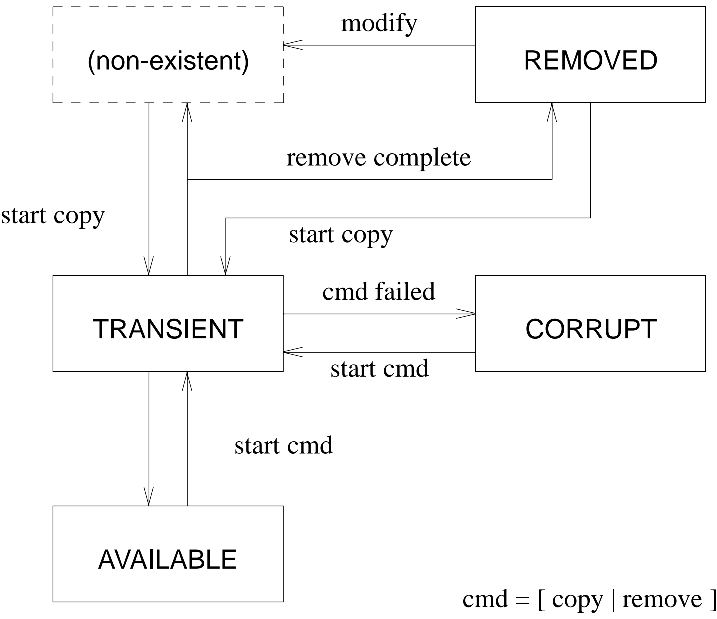

Figure B-5 Fileset State Transitions (Within Distributions)

Figure B-5 and Figure B-6 on page 202 show the state transition diagrams for Installed Software and Distributions.

It is clear that some vendors may want additional states. But allowing other values would make it problematic for any implementation trying to make decisions on how to deal with filesets with unrecognized states. An implementation may create an additional fileset attribute that would further modify the meaning of the attribute. For example, they may create an attribute called ***state_info*** and this attribute may have the value of `files_missing` when the ***state*** attribute is set to `corrupt`. There could be several valid values of this new attribute to describe various possibilities of a `corrupt` state. Of course, since this would be an implementation-specific extension, other implementations would not need to recognize this attribute or its semantics.

The default for updating, downdating or removing a fileset is to remove the information from the catalog. However, implementations may instead set these filesets to have a state of ***removed***.

The default behavior for filesets in the *removed* state is the same as for filesets that are ''non-existent''. Implementations that support the *removed* state should define extensions to the POSIX utilities providing operations on removed filesets. For example:

```
swlist -x show_removed=true
```

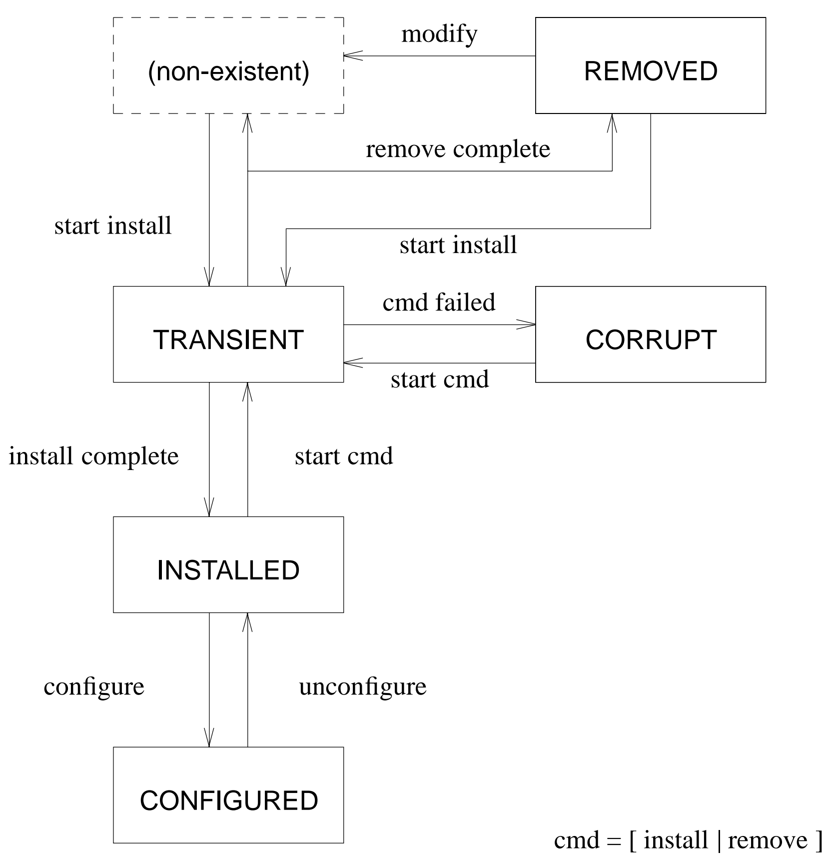

Figure B-6 Fileset State Transitions (Within Installed Software)

B.3.8 Exit Status

No additional rational is required under this heading.

B.3.9 Consequences of Errors

Since the utilities in this Software Administration specification operate on multiple software objects for multiple targets, the handling of error conditions (which is basically a policy decision) is complex. For example, should success be only all or nothing; or only, if all operations succeed for a specific host; or only, if all operations succeed for a specific product on all hosts? The type of error or errors causing such definitions of success or failure is implementation defined. Whether or not the user may specify policies regarding the way in which errors are handled is also implementation defined.

B.4 Software Administration Utilities

The following discussion on the approach of the 1387.2 developers to defining each software administration utility is presented in the same reference manual page style as that used in Chapter 4 for the actual definition of these utilities.

NAME

swask — Ask for user responses"

SYNOPSIS

No additional rationale is required under this heading.

DESCRIPTION

The purpose of this utility is to provide support for interactive requirements for software. By being able to execute these interactive scripts independently of *swinstall* and *swconfig*, it allows those utilities to still be scheduled for non-interactive execution.

The `request` scripts can be used to ask the administrator questions, or requests, where responses are needed by the software before installation or configuration.

The utility may be used to perform the following task:

- Answer the requests of software that has interactive customization needs.

The *swask* utility and `request` scripts are for software specific questions only. It does not provide any mechanism for implementation specific questions, although an implementation can choose to support implementation, site, or system-specific enhancements as normal implementation extensions.

OPTIONS

No additional rationale is required under this heading.

OPERANDS

No additional rationale is required under this heading.

EXTERNAL INFLUENCES

No additional rationale is required under this heading.

EXTERNAL EFFECTS

No additional rationale is required under this heading.

EXTENDED DESCRIPTION

This Software Administration specification has not included a naming convention or structure for storing per client response information. If a `request` script requires per client information, then it needs to store that information for all clients in the `response` file, and then locate the appropriate information during configuration.

This Software Administration specification provides a means for the `request` script to determine which clients are being requested, and for the `configure` script to determine that client. See Section 3.6.1 on page 56.

For example, a `request` script that requests per client keys is as follows:

```
set - $targets
echo 'keys="' > response
for i in $targets
do
    echo enter key for $i
    read j
    echo $i $j >> response
done
echo '"' >> response
```

and the corresponding `configure` script that looks up the correct key from the `response` file:

```
set - $keys
while [ -n $1 ]
do
    if [ "$1" = "$targets" ]
    then
        echo key is $2
        break
    fi
    shift; shift;
done
```

An implementation needs to ensure that any `response` files that already exist in the source or the catalog are copied to the **SW_CONTROL_DIRECTORY** before the `request` script is executed. The order of checks for `response` files allows for the following precedence:

- User input (if *ask=true*)
- Pre-existing response file
- Pre-existing client configuration
- Model response file (from source)

There are numerous ways to implement where the `request` scripts are executed and what **SW_CONTROL_DIRECTORY** is set to, for the command:

```
swask -s source -c catalog Foo.Bar
```

EXAMPLES

Example implementation A:

```
mkdir catalog/Foo/Bar
copy request script for Foo.Bar to catalog/Foo/Bar
copy any necessary response file to catalog/Foo/Bar
    if catalog/Foo/Bar/response exists, no action
    else if source/catalog/Foo/Bar/response exists, copy it
set SW_CONTROL_DIRECTORY=catalog/Foo/Bar
execute catalog/Foo/Bar/request
(remove catalog/Foo/Bar/request)
```

Example implementation B:

```
mkdir catalog/Foo/Bar /usr/tmp/aaaa43542/Foo/Bar
copy request script for Foo.Bar to /usr/tmp/aaaa43542/Foo/Bar
copy any necessary response file to catalog/Foo/Bar
    if catalog/Foo/Bar/response exists, no action
    else if source/catalog/Foo/Bar/response exists, copy it
set SW_CONTROL_DIRECTORY=catalog/Foo/Bar
execute /usr/tmp/aaaa43542/Foo/Bar/request
(remove /usr/tmp/aaaa43542/Foo/Bar)
```

Example implementation C:

```
mkdir /usr/tmp/aaaa43542
copy any necessary response file to /usr/tmp/aaaa43542
    catalog/Foo/Bar/response exists, copy it
    else if source/catalog/Foo/Bar/response exists, copy it
set SW_CONTROL_DIRECTORY=/usr/tmp/aaaa43542
execute source/catalog/Foo/Bar/request
mkdir catalog/Foo/Bar
cp /usr/tmp/aaaa43542/response catalog/Foo/Bar
(remove /usr/tmp/aaaa43542)
```

EXIT STATUS

No additional rationale is required under this heading.

CONSEQUENCES OF ERRORS

No additional rationale is required under this heading.

NAME

swconfig — Configure software

SYNOPSIS

No additional rationale is required under this heading.

DESCRIPTION

The purpose of configuration is to configure the host for the software, and configure the product for host-specific information. For example, software may need to modify the `/etc/rc` setup file, or the default environment set in `/etc/profile`. It may need to ensure that proper codewords are in place for that host, or do some compilations. Unconfiguration undoes these steps.

This utility may be used to perform the following tasks:

- Configuring software on target hosts that will actually be running the software
- Configuring independent of the remove and install utilities
- Configuring or unconfiguring hosts that share software from another host where the software is actually installed
- Reconfiguring when configuration failed, was deferred, or needs to be changed

OPTIONS

No additional rationale is required under this heading.

OPERANDS

No additional rationale is required under this heading.

EXTERNAL INFLUENCES

No additional rationale is required under this heading.

EXTERNAL EFFECTS

No additional rationale is required under this heading.

EXTENDED DESCRIPTION

When there is no script, the software is still transitioned to `configured` by *swconfig*. The state of the fileset without any configuration requirements is still changed to denote to the users of the software that the software is ready to use. Having one state for both software that requires configuration, and for software that does not, is easier than checking that all software that requires configuration is in the configured state, and that all software that does not require configuration is in the installed state.

If there are not sufficient (or any for that matter) responses in the `response` file, the `configure` script can log that further interaction is required and exit with a failure. This can prompt the user to execute the *swconfig* utility with *ask=true*.

The `request` script is executed at the manager role at the end of the selection phase, after the user has specified the software, but before analysis or execution begins on the target roles. The developers of this Software Administration specification considered defining a separate phase between the selection and analysis phases for *swconfig* and *swinstall*, but maintained the request steps as part of the selection phase for simplicity.

Reconfiguration may be useful when some system configuration has changed. This may include running with *ask=true* so the user can input different information.

There is the case where the `configure` script is not sufficient for configuring the software. If there is another configuration process that needs to be run, then this process should not change the state of the software to configured. After the other process is run, it can change the state to

configured using the *swmodify* utility. There are also situations where there can be multiple configurations of the same installed_software object. This Software Administration specification does not currently address this except by putting the burden on the software to manage the multiple configurations. This Software Administration specification does support the user rerunning the `configure` script each time a new configuration is needed. Using *swconfig* `-u` can likewise interact with the user to unconfigure one, but not all, of the configurations. For both of these cases, if the script exits with return code 3, the software does not transition to the `installed` (that is, unconfigured) state.

The `configure` scripts should also adhere to specific guidelines. For example, these scripts are only executed in the context of the host that the software will be running on so they are not as restrictive as customize scripts. However, in a diskless or NFS environment, they need to use file locking on any updates to shared files, as there may be multiple `configure` scripts operating at the same time on these shared files. The `configure` and `unconfigure` scripts need to be noninteractive, but may use the information in the `response` files generated by the `ask` script.

For diskless, cold install (initial OS install), and generally building an OS to a separate disk, *swconfig* can be automatically run after the system reboots to its real host to configure all unconfigured filesets.

This Software Administration specification does not define how file sharing, including diskless machines, should be implemented. However, separable configuration and installation steps provide the basic building blocks.

One possible file sharing solution involves each client having its own installed software catalog from which the shared software can be configured, and the `configured` state can be recorded. This catalog can be built by ‘‘link installing’’ the software; instead of loading files and running `preinstall` and `postinstall` scripts, link each product’s files to the client file system. Then, build the catalog information of this linked software as if it were installed and configure the client.

Another possible solution involves each client recording its configured state in a shared installed software catalog. In order to do this, the installed software could maintain a *configured_instances* attribute to hold a list of configured client names. Each client’s `configure` and `unconfigure` script could add or delete its name from this list.

These scripts could also control whether the installed software *state* attribute was changed from `installed` to `configured` via *swconfig*. If configuring, then the *reconfigure* option would need to be set to *true*. If unconfiguring, then the `unconfigure` script could exit with a return code of 3 (exclude) unless the *configured_instances* attribute was empty so that the installed software *state* would remain `configured`.

Update Capabilities

The 1387.2 standard simply runs a configure script during configuration as part of *swinstall* or as part of a separate *swconfig* (likewise it runs unconfigure as part of a *swremove* or as part of a separate *swconfig* `-u`).

The following is not covered by 1387.2, and must be performed in control scripts for distributions conformant with 1387.2:

- Removing obsolete files from previous revisions of a fileset (should be currently done in a configure script or a product postinstall and not a fileset postinstall script, due to recovery considerations)

- ‘‘User configurable files’’ (files that are shipped, but that the user can make changes to) can either be shipped in a ‘‘new configuration’’ location and conditionally copied during configure, or they can be shipped in-place but requiring a preinstall script to save off the existing file
- Any merging of existing user information and new software provided information is not supported by the utilities and must be done in a control script

Although the standard does not address file sharing, the separate configure phase is intended to allow multiple clients to run configure after the software has been installed on a file server.

If one version of software is installed and configured, and a newer version is installed in a different location, then *swinstall* will not automatically configure the second version. The *swconfig* utility needs to be run separately to:

- Either unconfigure the first version, then rerun to configure the new version
- By setting the ***allow_multiple_versions*** option to `true`, ***swconfig*** will attempt to configure the new version while still having the existing version in the configured state

Managing Updates

Recovery steps will account for the remove (‘‘*x*’’) file type, so this no longer needs to be done in configure scripts for shared files. Configure scripts still need to manipulate all private files.

For file sharing considerations, files should be determined to be ‘‘shared’’ or ‘‘private’’. In general, all read-only or binary files are considered shared and should be manipulated in the install execution phase (including the preinstall, file loading and postinstall steps). All user modifiable files are considered private should be manipulated in the configuration phase. In particular, prototypes of user modifiable files should be shipped as ‘‘shared’’ prototypes, and then copied or merged to a writable area (in the client’s view), during configure.

New files created by the configure script should be added to the installed software catalog using the swmodify utility.

Example:

```
swmodify -x files="newfile" PRODUCT.FILESET
```

Instead of merging old and new configuration files that may have been modified by users as well, it is better to design a system that is more updatable, such as a directory that contains system provided and user provided information in separate files. This allows newer versions to simply replace the files that they previously supplied.

EXAMPLES

No additional rationale is required under this heading.

EXIT STATUS

No additional rationale is required under this heading.

CONSEQUENCES OF ERRORS

No additional rationale is required under this heading.

NAME

swcopy — Copy distribution

SYNOPSIS

No additional rationale is required under this heading.

DESCRIPTION

This utility may be used to perform the following tasks:

- Copy software from one distribution to another
- Merge software from one distribution into another
- Copy software to a temporary distribution located to improve *swinstall* reliability or performance

OPTIONS

No additional rationale is required under this heading.

OPERANDS

No additional rationale is required under this heading.

EXTERNAL INFLUENCES

No additional rationale is required under this heading.

EXTERNAL EFFECTS

No additional rationale is required under this heading.

EXTENDED DESCRIPTION

The *swcopy* utility operates much like the *swinstall* utility. The key distinction between *swcopy* and *swinstall* is the way products are loaded. With *swcopy*, products are not installed for general use below the root directory. Instead, they are placed into a distribution, which can then act as a source for *swinstall*.

Update Capabilities

Unlike installation where products can not exist in the same location at the same time, every product that differs in any one of the revision, architecture or *vendor_tag* attributes is treated as a separate version of the product and does not update or overwrite any other version. Thus, when a newer revision of a product is copied into a depot, it does not in any way affect the older revision. As covered previously, swinstall will automatically choose the highest revision unless the revision explicitly specified.

When the same revision of a fileset within the same product version is being copied, it is considered a ‘‘recopy’’. The fileset is skipped unless the ‘‘recopy’’ option is set to `true`, just as is done with the *reinstall* option to *swinstall*.

Example:

```
swcopy -x recopy=true PRODUCT.FILESET,r=<same>
```

This will recopy all files in the fileset.

If the same product version has been repackaged so that attributes of the same product version (that is, same revision, architecture and vendor_tag), are different between the new product instance being copied in and the existing product instance, then the values of the new product are retained. The values of any attributes in the old product that are undefined in the new product are also retained.

An update within a depot is not clearly defined in 1387.2 (updates of filesets within existing product version are usually only needed during the development phase of a software product). Released products will generally have new revisions, or patches to a current revision.

EXAMPLES

Copy the *software_selections* listed in `/tmp/load.products` other default values defined in the `/var/adm/sw/defaults` file) as follows:

```
swcopy -f /tmp/load.products
```

Remove a product *Foo* from the distribution on the tape device `/dev/rct0` as follows:

```
swcopy -s /dev/rct0 \* @ /tmp/depot"
swremove -d Foo @ /tmp/depot"
swcopy -s /tmp/depot \* @ /dev/rct0"
```

Example of reinstall_files:

```
swcopy -x reinstall_files=false
```

Another example of reinstall_files:

```
swcopy -x reinstall_files=false reinstall_files_use_cksum=falsed
```

Examples of patch specification:

Example:

```
swcopy -x autoselect_patches=true  X11,r=6
```

Copy X11,r=6 software from one depot to another and also copy all patches for that base software.

Example:

```
swcopy -x autoselect_patches=true \
    -x patch_filter="*,c=critical"
```

Copy all software from a depot and also copy a filtered set of patches (those with a category of ‘‘critical’’), for the base software.

Example:

```
swcopy -x patch_match_target=true
```

Copy from a depot containing patches and also non-patch software, all patches for the base filesets which are already present in the target depot.

Example:

```
swcopy -x patch_match_target=true -x patch_filter="*,c=critical"
```

Copy from a depot containing patches and possibly also non-patch software, a filtered set of patches for the base filesets which are already present in the target depot.

EXIT STATUS

No additional rationale is required under this heading.

CONSEQUENCES OF ERRORS

No additional rationale is required under this heading.

NAME

swinstall — Install software

SYNOPSIS

No additional rationale is required under this heading.

DESCRIPTION

This utility may be used to perform the following tasks:

- Install operating system or application software, including configuration of operating system or application software as part of the install.
- Reinstall the exact same version as already installed.
- Update (or upgrade) operating system or application software to a higher revision, and downdate (or downgrade) operating system or application software to a lower revision.
- Apply software patches that have been packaged as standard software objects (filesets).
- Install additional versions of software when one already exists.

The install software task can have different connotations depending upon the life cycle stage of the software package. Figure B-7 illustrates the various state transitions that can occur during an installation of software.

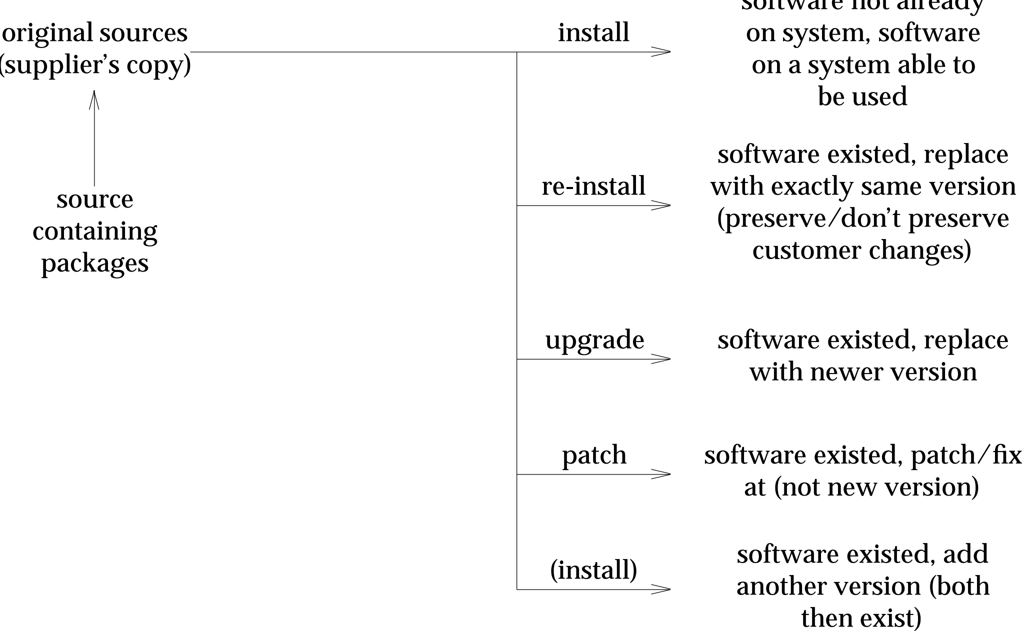

Figure B-7 Installation State Changes

These transitions comprise the set of install software tasks supported by this Software Administration specification.

The facilities provided by *swinstall* and *swcopy* are basic building blocks on which other function may be built. A few examples are shown below simply for the purpose of illustration. In the course of the illustration, various network sizes are given, but they are fictitious and supplied solely for illustration.

- Some users might consider *swinstall* to work well for doing remote installation to tens of machines. But with hundreds of machines, perhaps a better strategy would be to use *swcopy* to place a distribution on several servers for use in several parallel *swinstall* invocations. Also, this example could be cascaded for hierarchical operation with thousands or hundreds of thousands of machines.
- In addition, organizations that use special purpose software distribution programs could choose to use *swcopy* for that purpose, sending a copy of a distribution to some number of machines, pausing until assured that copies have arrived intact at all machines before proceeding, and finally causing *swinstall* to begin work on each machine at the same time.
- An installation process that makes efficient use of network resources might analyze the routing of distributions or files, discover that some traverse a common path before diverging, and cause only one copy to be sent over the common link using the *swcopy* utility.

OPTIONS

No additional rationale is required under this heading.

OPERANDS

No additional rationale is required under this heading.

EXTERNAL INFLUENCES

No additional rationale is required under this heading.

EXTERNAL EFFECTS

No additional rationale is required under this heading.

EXTENDED DESCRIPTION

The operations of the execution phase involve actual modification of the target file system and environment, and so from this point the utility is committed to the installation (in the sense that restoration to the state prior to load may not be possible).

Initially, *swinstall*, as well as *swremove*, would not act upon files in remote file systems. Although this was viewed as a good idea, it was removed. A vendor could check for local vs. remote file systems as a value-added feature, although it would require use of non-POSIX functionality. It should be noted that such a file system check involves policy, and thus should be overridable.

The normal use model is to install the latest revision of the product that matches the machine attributes of the targets. The default options fit this use model even when the targets specified have different uname attributes. As an example, assume the following command was issued:

```
swinstall -s source product1 @ host1  host2
```

Further, assume that host1 and host2 have different uname attributes and that the source contains instances of product1 that match the uname attributes of the two targets. The resulting install places the instance of product1 that matches the uname attributes of host1 onto host1, and the instance of product1 that matches the uname attributes of host2 onto host2.

The use of *allow_incompatible=true* is not the normal use model, and thus it is difficult to build in automatic filtering of selections. Thus, the user is required to uniquely specify which version is to be installed.

During disk space analysis, an implementation need only process paths with positive sizes in the `space` file to generate a maximum transient disk space requirement. Both space freed after successfully installing products when *autorecover=true* and negative size records can be used to generate a second and final disk space requirement.

How disk space analysis is implemented is undefined by 1387.2. When updating, each file size should be checked against any existing file at that location, on a per file system basis, unless rollback is enabled. In that case, the size of the existing files should be measured on the file system where they will reside until the install is committed.

Configuration occurs on hosts where the software is to be used, as opposed to the hosts that are serving software. For example, a host of one OS or architecture may be a file server of software for another OS or architecture. Configuration is where the software is configured specifically for or by the host, or vice versa, where the host is configured for or by the software. Examples are any compilation, modifying the host default `PATHNAME` for the software, or configuring the software for host-specific parameters. Contrast this to `preinstall` and `postinstall` scripts that should be written in a portable and relative (to the root directory) way since they may be executed in proxy. They are appropriate for operations that manage the files in the software that are to be served as an installation as opposed to being use as a configuration. Because the root directory is `/`, this is not an alternate root or proxy install, that is, the software is intended to be used on this host. When the root directory is not `/`, then a proxy install situation should be assumed, and the client of this installation instead should be rebooted and run the configuration. These client steps are undefined within this Software Administration specification. Since reboot is done only in non-proxy installations, the implementation of the ''target role'' that runs on the target architecture can execute the implementation-defined reboot process and then execute *swconfig*.

Reboot is executed after all filesets, kernel or otherwise, have been installed in order to avoid multiple reboots. This rationale is similar to a situation in kernel building. In that case, kernel building is only done after all filesets with the *is_kernel* attribute set to `true` are loaded in order to avoid multiple kernel builds.

The operating system underlying an implementation may require additional support for rebooting. For example, some systems may require intermediate reboots (reboot during the install process, with the install process continued after reboot). This, in turn, may require additional script return codes, or additional semantics for the *is_reboot* attribute. Clearly such additions are peculiar to the requirements of particular operating systems. Such additions should be documented by the implementation.

A file that is defined as */<product.directory>/<file>* is (by default) installed in the same location as follows:

/<product.directory>/<file>.

If a location was specified in the software selections, then the file is instead installed to the location as follows:

/<product.location>/<file>.

Additionally, if both an alternate root and a location was specified, the file is installed to the location as follows:

/<alternate_root>/<product.location>/<file>.

An example use for a `preinstall` script might be saving existing files before loading new ones. An example use for a `postinstall` script might be to clean up previously installed versions of a product, or to move a default file into place. These scripts are supplied by the vendor and are executed by the target role on each target. Note that the target role is not necessarily the system that will be using the software (the client role), and that multiple client roles may share the software from the same target role.

It is not sufficient to only state that the *postkernel* script is implementation defined. The name of the script, and also its location in the file system on each target host, should be known. Since

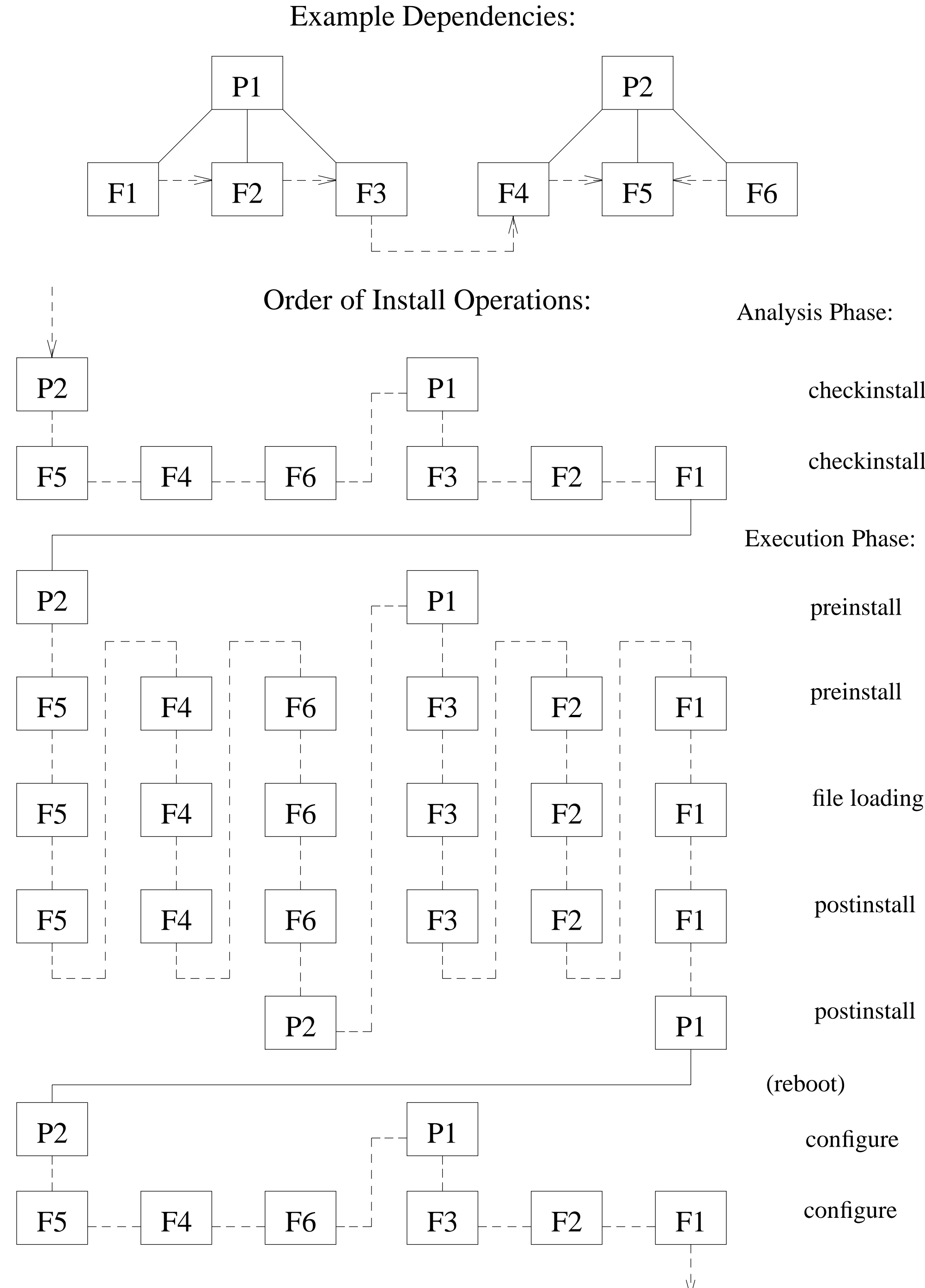

Figure B-8 Order of Install Operations

kernel filesets can be loaded for alien architectures (e.g., in diskless workstations), this script should also be aware of the effective root directory through the **SW_ROOT_DIRECTORY** environment variable. In general, all products with kernel filesets for a particular architecture should have the same *product*postkernel attribute defined. This script is contained within the product that contains the core operating system filesets. The implementation-defined default

path value for *postkernel* permits operating system implementation agreements that simplify the need for each product to define this attribute. A side effect of this product level attribute is that different products in the same distribution and same installation could conceivably have different values for the name of the script. In that case, multiple (probably redundant) kernel builds would occur and the order of fileset loads and kernel rebuilds would have to be specified.

The postkernel step after kernel filesets have been loaded is based on the HP-UX model for kernel building. This model supports both rebuilding of the kernel for OS updates, and insertion of an appropriate default kernel for new installations or installation in proxy. It also supports ensuring that a working kernel is in place before committing to the bulk of the install. For example, updating the HP-UX operating system to a new revision requires installing kernel related filesets and rebuilding the kernel successfully before loading the rest of the filesets. This ensures that the new kernel is operational (noted by the success of the *postkernel* script) before loading filesets whose executables will not run on the previous kernel. If the kernel build step was not insured to be successful, then you could have an unusable or even unbootable system.

When the installation is being done in proxy, the kernel build step should move a default new kernel into place instead of building a new kernel, since the architecture of the installed software may be different from the architecture of the server (the target role performing the installation). The script may check if it is an alternate root install by checking the **SW_ROOT_DIRECTORY** environment variable, and for the purpose of kernel building, it should take actions appropriate to a proxy install. Then, a new kernel containing the appropriate configuration (e.g., drivers) for the software installed may be built after or during the implementation specific configuration step (configuration occurs in the context of the system using the software).

Since the *postkernel* script is part of the target installed_software object, there is flexibility as to what actual steps are performed. For example, if a ‘‘cross-compiler’’ kernel building functionality existed in the target role, a new kernel could actually be built even in proxy.

There have been requests for removal of the required *postkernel* script return codes in *swinstall* on page 99, so making them undefined. It was concluded that it is important to keep them.

Update Capabilities

If there are multiple versions of software available in a depot (distribution), *swinstall* will choose the highest compatible revision of software if a revision, architecture or vendor_tag is not explicitly specified.

When installing a revision of a fileset, if there is currently a revision installed at the target location, then the install is considered an *update*, *reinstal* or *downdate*. In general, only updates proceed to execution phase; reinstalls and downdates are skipped.

During execution phase when updating, reinstalling or downdating a fileset, the catalog information for the existing fileset has its state set to ‘‘transient’’ during the course of an install, and then is removed or set to the ‘‘removed’’ state when the execution of that fileset completes.

By default, a file is installed into the location defined by its *path* attribute unless the product has been relocated. The only way to change the location of a file is change the root directory or the product *location*. This implies that any existing file at that location will be overwritten during the installation of the new fileset.

If the exact file (based on path, size, mtime and cksum attributes) is already installed, it is not reinstalled unless the user has specified that the filesets are being reinstalled. But, reinstall is defined as only applying to whether to reinstall a fileset with the same revision, not a file. This is vague when updates are considered (see below).

Example:

```
swinstall -x reinstall=true PRODUCT,r=1
```

When applying the POSIX defined recovery to update, if the ***autorecover*** option is set to `true`, then, if a fileset control script or file loading fails, then that fileset is removed, leaving the existing fileset in the state where it was previously. The next fileset is then attempted. If a product control script fails, then all filesets in the product that are being updated will be returned to their previous state.

Example:

```
swinstall -x autorecover=true PRODUCT,r=1
```

Downdate (or downgrade) is similarly defined, and behaves in much the same way as update. In general, software packages do not handle downdating very well (since the higher revisions of software have not been developed yet, the requirements for downgrading are not known). For this reason, the user must override the default behavior by setting the ‘‘allow_downdate’’ option to `true`.

Example:

```
swinstall -x allow_downdate=true PRODUCT,r=lower_revision
```

The administrator can also choose to install multiple versions of the same product (i.e. multiple revisions), by choosing different locations. The product ***is_locatable*** option must be `true` (the default). If a multiple version is being installed, swinstall will not Configure the new version. This is because the user needs to decide which version is the ‘‘current’’ version to be configured into the system, and then manually unconfigure one and configure the other.

The developers of this specification found ***autorecover_product*** preferrable to the POSIX ***autorecover*** that can result in a partially updated product.

The developers of this specification considered ***sparse*** upgrades good for marketing purposes (that is, publically available free upgrades for those that have the base, without having to give out the whole thing)

This specification provides capabilities for ‘‘recovery’’ to the previous state if an install fails (for example, due to a lost connection to the source or a failed preinstall or postinstall script). Since it is common practice for a new release (update or upgrade) of a product to be of better quality than a patch (or fix), functionality supporting manual rollback of an update is not included at this time. The administrator can manually rollback to a previous release by installing and managing multiple installed versions instead of performing an update, or by reinstalling the lower version (downdate or downgrade).

Examples:

```
swinstall -x reinstall=true -x reinstall_files=false
```

This reinstalls the fileset, but does not retransfer or reinstall any files that are already up to date.

```
swinstall -x reinstall_files=false PRODUCT,r=<newer>
```

This updates the product, but still checks each file before retransfer or reinstall.

Patch Operations

Several new attributes and options are used to implement these capabilities.

- In general, patches to a fileset can be either ‘‘superseding’’ (replacing previous patches to this fileset) or ‘‘point’’ patches (independent patches that can be installed at the same time, possibly with no side effects). This is controlled by the i *supersedes* attribute.
- In general, patches can be named after the product and fileset which they patch, or can have completely independent product and fileset names (for example, named after the defect number they fix). The fileset or filesets a patch applies to is defined by the *ancestor* attribute.
- In general, patches can be managed separately from regular software items. They are not ‘‘visible’’ to the casual user unless explicitly requested for install or listing. Selection for installation and listing is supported by the *category* attribute and special patch management options to the POSIX commands; explicit patches can also be included directly in POSIX software selections.

The following are examples of patch installation in same session as base product:

Example:

```
swinstall -x autoselect_patches=true X11
```

This is the default behavior for patch installation. All patches applying to software being installed (in this case X11) are selected.

Example:

```
swinstall -x autoselect_patches=true \
    -x patch_filter="*,c=critical" X11
```

These settings would cause all applicable patches that include the category ‘‘critical’’ to be selected and installed along with the selected software.

Example:

```
swinstall -x autoselect_patches=true \
    -x patch_filter="X11,c=critical"  BaseOS X11 Networking
```

These settings would cause only applicable patches for the product X11 that include the category ‘‘critical’’ to be selected and installed along with the other selected non-patch software ‘‘BaseOS’’ and ‘‘Networking’’.

Example:

```
swinstall -x autoselect_patches=false X11 \
    X11.Runtime,r=1.0.1 PH02-3425
```

This installs a product and two explicitly defined patches (showing different patch naming schemes). Note that the ‘‘autoselect_patches’’ defaults to `true`.

The following are examples of patch installation after base product is installed:

Example:

```
swinstall -x patch_match_target=true
```

These options will select the highest superseding patches in the depot that correspond to currently installed software.

Example:

```
swinstall -x patch_match_target=true -x patch_filter="X11"
```

These options will select the highest superseding patches in the depot that have an ancestor on system, and that have a product tag of ‘‘X11’’.

Example:

```
swinstall -x patch_match_target=true \
    -x patch_filter="X11,c=critical"  Networking.TCP,r=1.0.1
```

These options will select all critical patches with product tag X11 if their ancestors are installed and also will install an explicitly specified patch.

Example:

```
swinstall -x patch_match_target=true \
    -x patch_filter="*,c=critical"
```

These options will select all patches in the depot that correspond to currently installed software and that contain the category ‘‘critical’’.

The following is an example of Multiple criteria with patch filtering:

Example:

```
swinstall -x patch_match_target=true \
    -x patch_filter="X11,c=critical,c=test_level_4|hand_certified"
```

This will bring down any patches for X11 that have the category ‘‘critical’’ and either the category ‘‘test_level_4’’ OR the category ‘‘hand_certified’’.

The following is an example of Explicitly loading patches:

Example:

```
swinstall fix12312  Networking.TCP,r=1.0.1
```

This shows identification of a specific patch product (using one naming convention of naming the product for the defect it fixes) as well as explicitly identifying a patch fileset (using the other naming convention of naming patches after the filesets they patch).

The following is an example of saving patch files:

Example:

```
swinstall -x patch_save_files=true -x patch_match_target=true
```

FURTHER EXAMPLES

Install the C and Pascal products from the network source, *sw_server* as follows:

```
swinstall -s sw_server cc pascal
```

Install the C and Pascal products on some remote hosts as follows:

```
swinstall -s sw_server cc pascal @ hostA hostB hostC
```

Update the Omniback product from a CD-ROM mounted at `/cd` as follows:

```
swinstall -s /cd omniback
```

Install an incompatible version of Omniback as follows:

```
swinstall -x allow_incompatible=true omniback,a=foreign
```

Install all products from the cartridge tape `/dev/rct0` as follows:

```
swinstall -s /dev/rct0 \*
```

Install the `software_selections` listed in `/tmp/install.products` on the hosts listed in `/tmp/install.hosts` as follows:

```
swinstall -f /tmp/install.products -h /tmp/install.hosts
```

Example of match_target:

```
swinstall -s new_media -x match_target=true
```

Example of reinstall_files:

```
swinstall -x reinstall_files=false reinstall_files_use_cksum=false
```

Example of autorecover_product:

```
swinstall -x autorecover_product=true PRODUCT
```

EXIT STATUS

No additional rationale is required under this heading.

CONSEQUENCES OF ERRORS

No additional rationale is required under this heading.

NAME

swlist — List software

SYNOPSIS

No additional rationale is required under this heading.

DESCRIPTION

The *swlist* utility may be used to perform the following tasks:

- List distribution, installed_software, software and software_file objects
- List the attributes of each of the objects

It provides the means to read the catalog information. For installed_software, this information is stored in an implementation specific fashion. A read interface is needed to support portability between conforming implementations. In particular, an implementation may need to export current data for import into a new implementation.

A read interface should also support the objective of distributed administration by providing a standard interface for output of information about software being managed by any conforming implementation.

Finally, a vendor integrating into the standard software management environment needs a standard way to query the catalog about itself as well as other products, both from vendor-supplied scripts and the product execution itself.

The *swlist* command line interface meets these key objectives. There could be an additional requirement that a standard programmatic interface be defined for access to the catalog by related management applications, as well as software product executables. This objective can only be (reasonably) met by defining language specific or language independent interfaces, as well as the data structures needed to represent at least single software attributes. Alternatively, a standard object-oriented interface for retrieving objects and attributes from a standard administration catalog could be defined through which this Software Administration specification could support its defined software objects. This is not an objective of this Software Administration specification at this time.

Discussed, but not included, is a possible vendor extension for *swlist* to provide an interface into displaying the attributes of the defaults files for the various utilities. The syntax for listing an attribute would be

```
swlist -l <utility> -a <option>
```

OPTIONS

The output is the parsable form as it is in a known format, with attribute (keyword value) pairs. It is analogous to a POSIX.2 *ls* long listing. The undefined format should include at least the object *tag* attribute (for product, subproduct, fileset, control_file) or *path* attribute (for files).

Management of the host object (which contains the attributes for lists of distributions and installed_software objects) is undefined within this Software Administration specification. Still, and can be used to get lists of distributions and installed_software objects respectively on the host.

To list the files in a fileset object, then no option is needed. But, to list both the files and control_files, then both and are needed. Likewise, and can be used together to list both the subproduct and fileset objects contained in a product.

If the `-v` is not specified, a typical implementation might list all attributes specified on one line per object.

Both the `-l` and `-a` options can be used to obtain lists of contained objects where `-l` returns available or currently installed objects and `-a` returns all objects defined in the original distribution. Only the `-l` option can be used to obtain attributes of those contained objects. Note that the keyword for `-a` ***attribute*** is just the plural of the keyword for `-l` ***level*** . For example:

```
swlist -a control_files <fileset>
```

lists the defined control_file tags for the fileset, while:

```
swlist -l control_file -a result <fileset>
```

lists the ***result*** attribute of each control_file object in the fileset.

Note that the `-l` option could be extended to one more level, host, that would list all distributions on the host.

Both the ***targets*** operand and the `[-s` ***source***`]` option were considered for administrator convenience, but the latter was removed from this Software Administration specification. The former is used to list software available from a distribution that is being used as a target. The latter was possible for listing software available from a distribution that is being used as a source with other commands; however, it can also be accessed as a target.

In addition to all attribute names defined in Software Administration specification, the following three additional items are supported by the `-a` ***attribute*** option: `software_spec`, `mod_date` and `create_date`.

The `software_spec` item is supported in order for users of ***swlist*** to generate the exact conformant `software_spec` needed to uniquely identify a software object. This was included because creating a `software_spec` from the correct version distinguishing attributes is moderately difficult.

The `mod_date` and `create_date` items are intended for use by administrators to easily see when the software object was first installed or available, or when it was last modified. The format of these strings should be time zone and locale dependent, such as:

```
Jan 14 1993 4:30 PM
```

In addition, `software_spec` , `date`, and `create_date` should not be stored in any software definition file. If they are found there, they will be ignored. Whether they are implemented as attributes, methods on existing attributes, or some other method, it does not matter. They may be specified like any other attributes for administrator convenience.

OPERANDS

No additional rationale is required under this heading.

EXTERNAL INFLUENCES

The ***one_liner*** extended option is used to create default output for the novice user, as well as for the experienced user. A typical implementation might list all attributes in the ***one_liner*** on one line per object. The ***tag*** and ***path*** attribute behavior also applies if the ***one_liner*** is undefined or set to no attributes. If the ***tag*** and ***path*** are really not desired in the output, then the `-a` ***attribute*** option may be used.

More advanced support for formatting output, such as a ***format*** option using ***printf***() syntax, was discussed, but it was deemed that this is a reasonable area for vendor extension, and not necessary for this Software Administration specification given the `-v` parsable format.

EXTERNAL EFFECTS

No additional rationale is required under this heading.

EXTENDED DESCRIPTION

Update Capabilities

When a product is partially updated (one or more, but not all, filesets have been updated by filesets from a different version of the product), then two product versions exist on the system, each partially complete. The *swlist* utility will show both.

Example:

```
swlist -l fileset -x one_liner="revision  title" SW-DIST

# SW-DIST            B.10.10  HP-UX Software Distributor
  SW-DIST.SD-AGENT   B.10.10  HP-UX Software Distributor agent
# SW-DIST            B.10.20  HP-UX Software Distributor
  SW-DIST.SD-CMDS    B.10.20  HP-UX Software Distributor commands
```

Once the product is completely updated, only the newer version is listed.

Example:

```
swlist -l fileset -a revision -a title SW-DIST

# SW-DIST            B.10.20  HP-UX Software Distributor
  SW-DIST.SD-AGENT   B.10.20  HP-UX Software Distributor agent
  SW-DIST.SD-CMDS    B.10.20  HP-UX Software Distributor commands
```

Patch

Filtering according to category will be supported. The use of category filtering will allow easy selection of all patch software objects. Software objects with the *is_patch* attribute set to true have the built-in category of ''patch''. The category ''patch'' is reserved for this use.

This can be used to list available patches, and patches with a certain name.

The fileset attribute *applied_patches* is automatically updated to include any patches that have been installed and applied to it, due to it being in the patch fileset's ancestor list. This can include patches that perhaps do not share the same product tag as the base filesets (ancestors) they patch.

Any fileset that has been patched has an *applied_patches* attribute listing all of the patches that have been applied to it.

Listing what products and filesets a patch applies to can be generated by listing the *ancestor* attribute. A list of what patches a patch superseded can be generated by listing the *supersedes* attribute.

EXAMPLES

List all distributions on a host:

```
swlist -d -l distribution @ host
```

List all products in the default distribution:

```
swlist -d
```

List all files in an installed fileset:

```
swlist Product.Fileset
swlist -a files Product.Fileset
swlist -l file Product.Fileset
```

List all files and control_files in a fileset, in `INFO` file format:

```
swlist -v -l file -l control_file Product.Fileset
```

List the states of control_files in product:

```
swlist -a state -l control_file Product
```

List the definition of the contents of a bundle in the default distribution:

```
swlist -d -a products -a bundles Bundle
```

Dump an installed_software catalog:

```
swlist -l file -l control_file -l bundle -l subproduct -c
    exported_catalog
```

List a fileset software object into the software definition file format:

```
swlist -l fileset -v -c exported_catalog Product.Fileset
```

List a product's copyright:

```
swlist -a copyright Product
```

List the values of all product and fileset software: ***tag***, ***revision***, and ***title*** attributes from the default distribution:

```
swlist -d -l fileset -x one_liner='tag revision title'
```

Example of listing sparse filesets:

```
swlist -l fileset -a revision -a is_sparse PRODUCT,r=10.0

# PRODUCT              10.0
  PRODUCT.FILESET       1.0    false
  PRODUCT.FILESET       2.0    true
```

Examples of patch listing:

Example:

```
swlist -d -l product -l bundle -a software_spec *,c=patch
    PH003-23245,r=1.0,a=,v=
    PH056-45545,r=1.0,a=,v=
    X11,r=5.00.01,a=FLUX,v=FLB
```

This lists all products and bundles in a depot that have the ***is_patch*** attribute set to `true`.

Example:

```
swlist -l fileset -a software_spec X11,c=patch

X11.Runtime,r=5.00.01,a=FLUX,v=FLB,fr=1.0.1
```

This lists all patch filesets in the product X11 (but only patches that have the product tag X11).

Example of listing applied patches:

Example:

```
swlist -l fileset -a applied_patches X11

X11.Runtime
X11.Runtime,r=1.0,a=FLUX,v=FLB,fr=1.0.1
X11.Manuals
X11.Libraries    PH056-45545.PH056-45545,r=1.0,a=FLUX,v=FLB
```

This shows patches with matching and non-matching tags.

Example of listing patch categories:

Example:

```
swlist -d -l category -s /CD
```

`critical`	Patches that fix system hangs or data corruption.
`S747_upgrade`	Patches that are needed to upgrade to an S747.
`security_patch`	Patches affecting system security.

The *swlist* command lists the categories defined in the depot mounted at `/CD`. The attributes of the category objects are controlled by the POSIX one_liner default (in this case set to ''tag title'').

Example:

```
swlist -a description -l category critical
```

List a particular attribute of a category object identifed by the tag ''critical''.

Note: 1387.2 could also use a `-l` *vendor*.

EXIT STATUS

No additional rationale is required under this heading.

CONSEQUENCES OF ERRORS

No additional rationale is required under this heading.

NAME

swmodify — Modify software catalog

SYNOPSIS

No additional rationale is required under this heading.

DESCRIPTION

The *swmodify* command provides the write interface to the catalog information. For installed software this may be in an implementation specific manner. A write interface is needed to support portability between conforming implementations. In particular, an implementation may need to import current data dumped from a previously used implementation.

Another objective of this Software Administration specification to provide administrator portability. A write interface provides a standard way for administrators to integrate software initially administered (installed) using other tools besides those defined in this Software Administration specification.

Finally, there needs to be a standard way for a software vendor to access and modify attributes in the catalog belonging to their product from their vendor-defined scripts, or from the execution of the product itself.

The *swmodify* command line interface meets these key objectives. There could be an additional requirement that a standard programmatic interface be defined for access to the catalog by the target role. The objective here would be to be able to link in different conforming catalog implementations with the conforming target role. This objective can only be (reasonably) met by defining a programmatic interface, as well as the data structures needed to represent all software objects and attributes. Alternatively, a standard object-oriented interface for setting and committing objects to a standard administration catalog could be defined through which this Software Administration specification could support its defined software objects. This is not an objective of this Software Administration specification at this time. This utility may be used to perform the following tasks:

- Modify or fix attributes at all levels of a distribution or installed_software object.
- Add and delete software objects from within scripts or other management interfaces.
- Add and delete software file objects from scripts or other management interfaces. This is similar to *installf* in SVR4.
- Convert the format of installed_software databases such as when one database technology is being replaced by another.

A proposal was discussed to extend *swmodify* to provide an interface into modifying the attributes of the defaults files for the various utilities. The syntax for modifying an attribute would be

```
swmodify -l <utility> -a <option>=<value>
```

OPTIONS

No additional rationale is required under this heading.

OPERANDS

When an exported_catalog is specified (a file in the software definition format or a directory in the exported catalog structure), any software selections apply to the exported_catalog, not the target. If the software from the exported_catalog matches more than one software object in the target, those modifications are not applied.

If an exported_catalog is not specified, the software selections apply just to the target. If they resolve to more than one software object, those modifications are not applied.

EXTERNAL INFLUENCES

The *files* option provides a convenient interface for adding to the catalog files that were created by control scripts, without having to create an intermediate input file. This provides function equivalent to that of the SVR4 *installf* command. For example:

```
cp   $SW_ROOT_DIRECTORY/$SW_LOCATION/lib/default \
     $SW_ROOT_DIRECTORY/etc/file
swmodify -x files=$SW_ROOT_DIRECTORY/etc/file  \
         Product.Fileset,l=$SW_LOCATION
```

EXTERNAL EFFECTS

No additional rationale is required under this heading.

EXTENDED DESCRIPTION

No additional rationale is required under this heading.

EXAMPLES

Reload an installed_software catalog:

```
swmodify -c swlist_output_dir
```

Add a software object defined in the software definition file format:

```
swmodify -c swmodify_input_file
```

Delete an obsolete fileset:

```
swmodify -u Product.Fileset
```

Delete three files from a `configure` script:

```
swmodify -u -x files="file1 file2 file3"  Product.Fileset
```

Modify a distribution level attribute:

```
swmodify -d -a name=$NAME
```

Committing a patch:

```
swmodify -x patch_commit=true X11.Runtime,r=1.0.8
```

The patch shown above is now committed and can not be rolled back.

Note: *patch_commit* needs to be able to commit all patches back to the base.

EXIT STATUS

No additional rationale is required under this heading.

CONSEQUENCES OF ERRORS

No additional rationale is required under this heading.

NAME

swpackage — Package distribution

SYNOPSIS

No additional rationale is required under this heading.

DESCRIPTION

This utility may be used to perform the following tasks:

- Package software into a new or existing distribution directory on the local host.
- Package software into a new serial distribution that may be a character-special file representing a DDS, nine-track tape, cartridge tape, or a regular file, within which the POSIX.1 extended *tar* or extended *cpio* archive is stored.

Both outputs use the same software packaging layout, but in different formats (directory versus serial).

OPTIONS

No additional rationale is required under this heading.

OPERANDS

No additional rationale is required under this heading.

EXTERNAL INFLUENCES

No additional rationale is required under this heading.

EXTERNAL EFFECTS

No additional rationale is required under this heading.

EXTENDED DESCRIPTION

A target distribution may only be the first step in creating a CD-ROM, either as a serial or directory distribution. There is no special support provided by *swpackage* for the creation of CD-ROM.

Distribution tapes are created in POSIX.1 extended *cpio* or extended *tar* format. To create the tape, *swpackage* first builds the products into a temporary distribution. (It is removed when *swpackage* completes.) To conserve space, all files may exist as references to the real source files. After the distribution is constructed, *swpackage* then archives it, along with the real files, onto the tape device.

The *swpackage* command supports two methods for making additions and/or modifications to an existing distribution:

1. Modify the PSF used to package some or all of the products in the existing distribution. Then invoke *swpackage* and specify the appropriate *software_selections* on the command line. Each specified software selection should correspond to a modified product, subproduct, or fileset definition within the PSF. If new filesets are being added to an existing product, *swpackage* will identify the product and add the filesets. If product, subproduct, or fileset attributes are being modified, *swpackage* will match them and do the replacements.
2. Create a new delta PSF for the products being modified. Send this delta file as the source PSF, and *swpackage* will repackage all of the specified products. The smallest unit that can be changed is the fileset. If a file within a fileset is added, modified, or deleted, the new fileset would replace the existing fileset. Enough product information needs to be given to correctly identify the product in which this fileset is located. When *swpackage* reads the delta PSF, it parses and deals only with the differences.

As the delta PSF comes closer to the original PSF, the repackage process will take as much time or more as the original packaging.

If a modified product, subproduct, or fileset specification redefines any attributes, the new attribute values will replace the existing values.

EXAMPLES

Package the products specified in `/develop/product_list` onto a distribution located at the default location:

```
swpackage -s /develop/product_list
```

Package the same products onto the distribution located at `/develop/cd`, from where they can printed onto a master CD-ROM as follows:

```
swpackage -x media_type=directory -d /develop/cd2
          -s /develop/product_list
```

Create a tape media on a 1300 MB DDS tape loaded at `/dev/dat` as follows:

```
swpackage -x media_type=serial -x media_capacity=1300 -d /dev/dat2
          -s /develop/product_list
```

EXIT STATUS

No additional rationale is required under this heading.

CONSEQUENCES OF ERRORS

No additional rationale is required under this heading.

NAME

swremove — Remove software

SYNOPSIS

No additional rationale is required under this heading.

DESCRIPTION

This utility may be used to perform the following tasks:

- Remove installed software.
- Remove software from a distribution.
- Remove software patches that have been packaged as standard software objects (filesets).

OPTIONS

No additional rationale is required under this heading.

OPERANDS

No additional rationale is required under this heading.

EXTERNAL INFLUENCES

No additional rationale is required under this heading.

EXTERNAL EFFECTS

No additional rationale is required under this heading.

EXTENDED DESCRIPTION

The remove software utility is a reversal of the install software utility. The stages, such as check remote, pre-remove customization, etc., are identical in purpose to the counterpart stage of the *swinstall* utility.

The *swinstall* utility automatically installs all filesets that a selected fileset depends on, but *swremove* needs to be explicitly told to remove filesets that depend on a selected fileset. This may be changed by setting *autoselect_dependents*true. The *swremove* utility has this default because it is easier to remove additional things if they are not wanted (that is, remove the dependencies) than it is to restore something that was erroneously removed.

When removing software objects, but not a bundle or subproduct that refer to those objects, the standard specifies the behavior as implementation-defined. Consensus on this matter was not achieved. An implementation may choose to remove the object, to warn the user and remove the object, or make this case an error and not remove this object. If this is an error, then the user should have a way to override this behavior.

The general consensus was that removing part of a bundle should be a warning or an error, and removing part of a subproduct should be a note or a warning. A bundle will not get installed unless explicitly specified. So, removing part of another installed bundle is not necessarily desirable. On the other hand, a subproduct may get installed as a result of a product being installed. Also, if there is a good reason why the subproduct needs to be complete (that is, there is a dependency on it), then an implementation may generate an error or warning based on dependency checks.

An example use of an `unconfigure` script is to perform an orderly shutdown of an application in preparation for removing it. The `unconfigure` script is a script or program supplied by the vendor. It is executed by the client role on each target host.

Originally, the behavior of *swremove* was to remove empty directories not referenced by other products. The ability to have multiple catalogs made this impractical. Since this seemed overly restrictive, the behavior with respect to directories is now implementation defined.

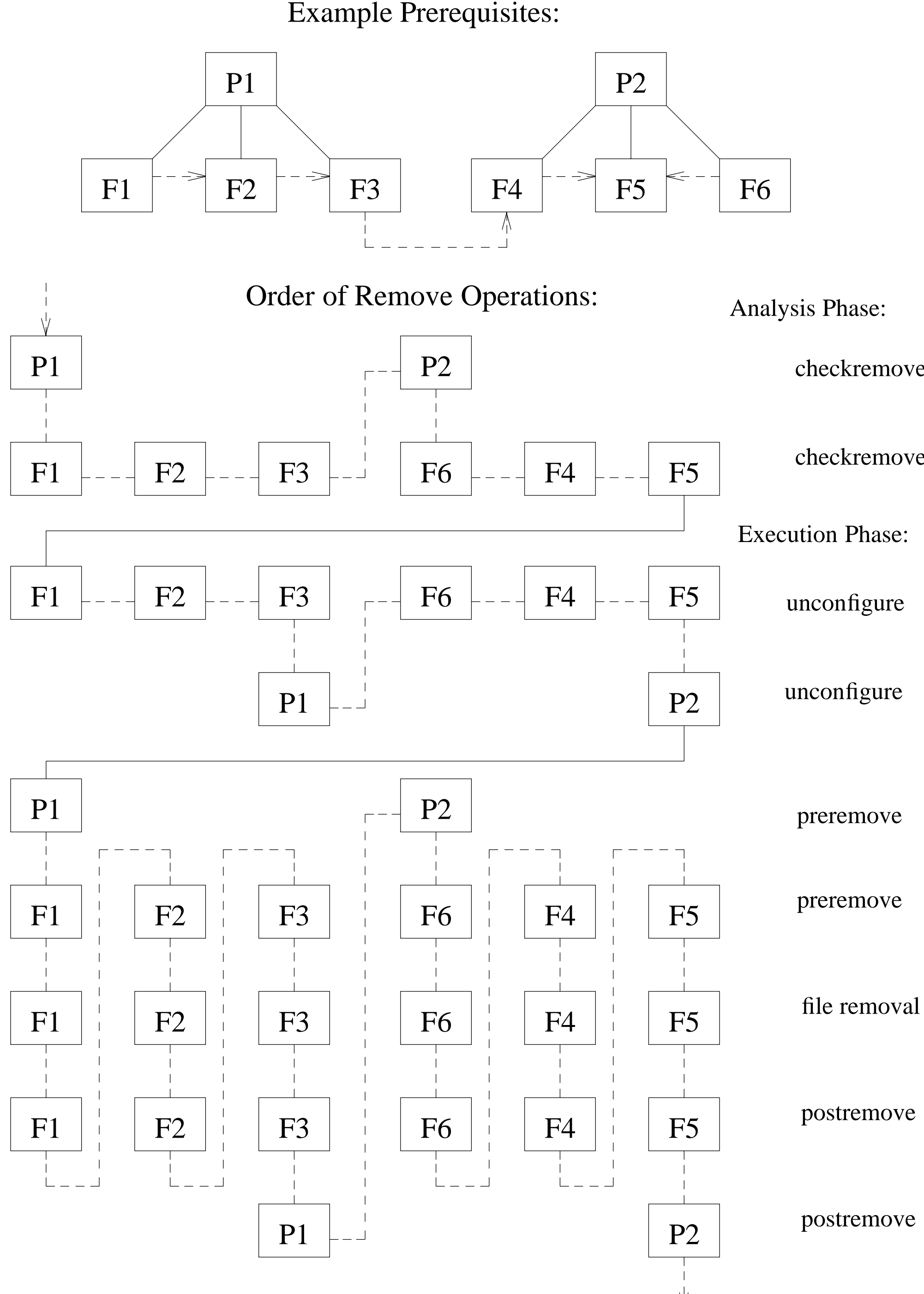

Figure B-9 Order of Remove Operations

Note, however, that unremoved directories should still be logged.

The following is an example of rolling back a patch:

Example:

```
swremove X11.Runtime,r=1.0.8
```

This removes the patch `X11.Runtime,r=1.0.8` and restores the saved files.

EXAMPLES

Remove the C and Pascal products as follows:

```
swremove cc pascal
```

Remove the C and Pascal products stored in the distribution */var/spool/sw* as follows:

```
swremove -d cc pascal @ /var/spool/sw
```

Remove the *Green* and *Blue* subproducts from the **Foo** product, and write detailed messages to the stdout/stderr, as follows:

```
swremove -x verbose=2 Foo.Green Foo.Blue
```

Preview what would happen if the *Green* and *Blue* subproducts were removed from the *Foo* product as follows:

```
swremove -p Foo.Green Foo.Blue
```

Remove the *software_selections* listed in */tmp/remove.list* as follows:

```
swremove -f /tmp/remove.list
```

EXIT STATUS

No additional rationale is required under this heading.

CONSEQUENCES OF ERRORS

No additional rationale is required under this heading.

NAME

swverify — Verify software

SYNOPSIS

No additional rationale is required under this heading.

DESCRIPTION

This utility may be used to perform the following tasks:

- Verify the installed software, including dependencies, file attributes, and running `verify` scripts.
- Verify the distribution software, including dependencies, and file contents attributes.
- Fix installed software information including running `fix scripts.`

OPTIONS

No additional rationale is required under this heading.

OPERANDS

No additional rationale is required under this heading.

EXTERNAL INFLUENCES

File system permissions cannot be verified in distributions since there is no requirement that distribution files have the permissions set to those defined for them. The *swinstall* utility sets the permissions based on the attributes defined, not the existing permissions. It is expected that the default permissions an implementation uses for files within distributions is the same as the attributes defined. However, there are some security considerations, as well as media considerations, such as for CD-ROMs, that would make those permissions different.

Note that the `verify` script is free of side effects.

Originally, the options with the *check_* prefix had command line options. However, it was felt so many option letters confused the command line. In addition, the semantics of which checks to perform were difficult to specify. It was felt that by adding these options to the options file, a system could be configured to perform those set of checks to be used in the normal case. When a different set of checks are desired, a user could override specific checks with the `-x` *option=value* option or specify a different options file with the `-X` *options_file* option.

EXTERNAL EFFECTS

No additional rationale is required under this heading.

EXTENDED DESCRIPTION

A `verify` script is used to ensure that the configuration of the software is correct. Possible vendor-specific operations for a `verify` script include the following:

- Determination of active or inactive state of the product.
- Check for corruption (correctness) of product configuration files (writable files).
- Check for correct and incorrect configuration of the product into the OS platform, services, or configuration files.
- Check licensing situation.

Note also that if a `preinstall` or `postinstall` script changes attributes of a file during install, then the *swmodify* command should be used to adjust those attributes in the installed_software catalog. Otherwise, file level verify will likely fail.

A `fix` script is used to attempt to fix the installed software. Possible vendor-specific operations for a `fix` script include similar operations to rerunning a `preinstall` or `postinstall`

script. Operations similar to rerunning a `configure` script are also possible if the state of the software is configured.

Like the file types of `d` (directory), and `s` (symlink), ‘‘*swverify* `-F`’’ (fix) will remove any files with type *x* (delete file) during execute phase.

Note: ‘‘*swverify* `-F`’’ does not fix size, mtime or cksum.

EXAMPLES

No additional rationale is required under this heading.

EXIT STATUS

No additional rationale is required under this heading.

CONSEQUENCES OF ERRORS

No additional rationale is required under this heading.

B.5 Software Packaging Layout

Software to be distributed according to this Software Administration specification is represented in a form defined as the software packaging layout. The software packaging layout provides a standard for representing software using the exported form of the software catalog, and a structure for the actual software files.

The reason for specifying a standard software packaging layout is to allow for interoperable media. In other words, any conforming distribution may be read by any conforming implementation.

This layout does not dictate how the software files are actually located in the file system. The layout provides a portable means to store and distribute files for any file organization. Installing could involve a transformation from a canonical form to one that conforms to the requirements of the architecture of the machine on which the software is to run.

The software catalog refers to the information (metadata) that describes software objects in a particular distribution or installed software object. Each catalog describes exactly one software_collection (installed_software object or distribution).

The information in the software catalog may be created, altered, and removed using the utilities defined in this Software Administration specification. A system administrator may use *swlist*, for example, to obtain a list of installed software products, the shell scripts used for installing a product, or the files contained within a product.

Each system has a default catalog for installed_software. This catalog provides information on the products installed for use on that system. Permission to modify the catalog may be restricted to the system administrator. Since there are times when users without special privileges may wish to use software administration tools to install and maintain software in their parts of the file store, additional installed_software objects, in separate software catalogs, may be created and acted upon by any user having appropriate authority.

The way that installed_software catalog information is stored (whether kept in text files or databases, whether distributed or centralized), is undefined within this Software Administration specification.

B.5.1 Directory Structure

An implementation may want to support several levels of INDEX files. The global INDEX file shall contain the correct information, and other lower level INDEX files may be used for convenience of the implementation when building the global INDEX. The following INDEX files, when simply concatenated together, form the following global INDEX:

- catalog/dfiles/INDEX
  A distribution INDEX contains the definition of the attributes set for the distribution itself and the bundle definitions.
- catalog/<***product_control_directory***>/pfiles/INDEX
  A product INDEX contains the definition of the product and subproduct attributes for the product object that *product_control_directory* represents.
- catalog/<***product_control_directory***>/<***fileset_control_directory***>/INDEX
  A fileset INDEX contains the definition of the fileset attributes for the fileset object that *fileset_control_directory* represents.

An example of product control directory name follows. If a product with the *tag* attribute value of SDU is added to a distribution, and no other objects with the same value of *tag* exist, the *product_control_directory* for storing the product object is SDU/. If another product object with the *tag* SDU is added to the distribution, the *product_control_directory* for that product object is SDU.2/. The first SDU product was assigned the value of 1 for its *product*instance_id.

B.5.2 Software Definition File Format

Through the software definition files contained in this Software Administration specification, an exported catalog structure representation is defined within the context of a distribution. This is necessary to have interoperable media. It is also used to define an exported catalog structure representation for installed_software objects. This is necessary for *swlist* output.

In the PSF, attributes may be undefined or set explicitly unless their default value is none. Most attributes have a default value. Default values are logically required for attributes that may be optionally specified in the PSF.

Most Boolean attributes default to false, such as *is_reboot*, *is_kernel*, and *is_locatable*. Instead of defaulting all Boolean attributes to false through clever wording of the attribute tag, some default to true, such as *is_locatable*.

The **distribution** keywords and the *layout_version* specification should not be changed in future amendments to this Software Administration specification so an implementation can determine the version of the layout being used.

In the PSF, the < (less than) is a token for unambiguous parsing, but it does not imply that the file contents should be included (that is, file re-direction). The alternative is to take the file from that path on the development system and store it as a control_file (for example,for example, a *copyright* control_file containing that attribute).

Vendor-defined attributes include any attribute that has a keyword not defined by this Software Administration specification. Since the value of any vendor-defined attribute is interpreted simply as a string, the original value entered in the PSF is preserved through packaging and copy operations. The value is thus preserved until the package is finally installed. During installation, *swinstall* may be affected by vendor-defined attributes in the INDEX or INFO files and may choose to preserve or not preserve the associated values for such keywords. The *swinstall* utility is only required to preserve in the catalog those values that correspond to the keywords defined in this Software Administration specification. Refer also to portions of the extended description for the *swinstall* utility.

Since device file creation is not possible in a portable way (that is, not included in POSIX.1 or POSIX.2), only a `configure` script that is running on the target system can reliably create devices files. POSIX.1 does define the ‘‘syntax’’ of *major* and *minor*, so it is possible for the `configure` script to add these attributes via *swmodify* (and for *swmodify* to look up their values through POSIX.1 interfaces). This in turn allows *swverify* to verify that these are correct.

The extended file definition syntaxes provide a flexible interface to individual software vendor development and build systems. These build systems may contain files to be packaged existing anywhere from intermingled with source files to a target build tree containing the files in the locations they would be in after installation.

- The directory mapping is intended for use with target build trees where all files are below a build root. The build root maps to some directory in the installed location such as `/` or some subdirectory like the *product_control_directory*.
- Recursive (implicit) file specification can be used with the previous instead of explicitly defining each file. In this case the mapped directory shall contain only the files from that particular fileset, and shall map to some directory (for example, `/`) on the installed system.
- Explicit file specification is useful for build systems where files are not organized in the same way they are to be installed, and for build systems that want the control of listing each file explicitly (implicit specification could possibly lead to unwanted files being packaged).
- Packaged files, by default, take the permissions of the files being packaged. Specification of permissions override the permissions of the files being packaged. Alternatively, each file may have its permissions explicitly defined.

The following examples illustrate the use of the **directory** and **file** keywords:

1. Include all files under `/build/s/`, to have a *product.directory* as `/opt/sw`:

```
directory  /build/s /opt/sw
file  *
```

2. Include only certain files under `/build/s/`, to be rooted under `/opt/sw`:

```
directory  /build/s /opt/sw
file  bin/swinstall
file  -t h  bin/swinstall bin/swcopy

file  -t d  lib/nls
file  nls/swinstall.cat lib/nls/swinstall.cat

file  data/swinstall.defaults newconfig/defaults/swinstall
file  data/swinstall.defaults /var/adm/sw/defaults/swinstall
```

3. Explicitly list files, no directory mapping specified:

```
file  /build/s/bin/swinstall /opt/sw/bin/swinstall
file  -t h  /opt/sw/bin/swinstall /opt/sw/bin/swcopy

file  -t d  /opt/sw/lib/nls/
file  /build/s/nls/swinstall.cat /opt/sw/lib/swinstall.cat
```

4. Use all specification types to include files:

```
directory  /build/s /opt/sw
file  *
file  bin/swinstall
file  -t h  bin/swinstall bin/swcopy

file  data/swinstall.defaults newconfig/defaults/swinstall
file  data/swinstall.defaults /var/adm/sw/defaults/swinstall
```

The following examples illustrate the use of the **file_permissions** keyword:

1. Set a read-only 444 mode for all file objects (requires override for every executable file and directory):

```
file_permissions  -m 444
```

2. Set a read mode for non-executable files, and a read/execute mode for executable files and for directories:

```
file_permissions  -u 222
```

3. Set the same mode defaults, plus an owner and group:

```
file_permissions  -u 222 -o bin -g bin
```

4. Set the same mode defaults, plus a uid and gid:

```
file_permissions  -u 222 -o 2 -g 2
```

5. Set the owner write permission (in addition to the above):

```
file_permissions  -u 022 -o 2 -g 2
```

If the user defines no **file_permissions**, *swpackage* uses the default value:

```
file_permissions -u 000
```

for destination file objects based on existing source files. This means that the mode, owner/uid, and group/gid are set based on the source file, unless specific overrides are specified for a destination file.

The default value:

```
file_permissions -u 000 -o bin,2 -g bin,2
```

is used by *swpackage* for destination file objects being created (not based on existing source files).

If neither the path nor any attributes is specified, then the suggested default values of the attributes are:

```
-m 0777 -o bin,2 -g bin,2
```

These are only suggestions, since the default values in this case are implementation defined.

The `space` file is used to account for disk space used (positive size) or freed (negative size) by control script execution. Negative sizes may be used by a more sophisticated disk space analysis that accounts for transient as well as final disk space. For example, if a fileset has a `configure` script that creates a 200 KB file `/usr/foo/data/file`, it can include a `space` file with the entry:

```
/usr/foo/data 200000
```

This will tell the implementation to add 200 KB to the disk space requirements. If the `configure` script moves a 35 KB file from `/etc/file` to `/sbin/bin`, it could include the entries:

```
/etc/file -35000
/sbin/file +35000
```

An implementation may choose to ignore the negative entry, producing a ''worst case'' disk space requirement. Alternatively, the implementation may choose to list the ''worst case'' requirement (ignoring the negative entry, reflecting the transient disk space required), and then the final disk space (accounting for the negative entry). Finally, if a `configure` script created a temporary 5 MB file under `/tmp`, it could add the entries:

```
/tmp  5000000
/tmp -5000000
```

The following are examples for ''update packaging'':

Example of type (x):

```
file -t x /oldfile
```

This adds a remove file to the fileset definition. It gives the software packager explicit control over which files to remove (useful if any of the files from previous revisions were user configurable).

Example of ancestor

```
product
    tag NEWPROD
fileset
    tag NEWFILESET
    ancestor  OLDPROD.OLDFILESET,r=10.0
```

This designates NEWPROD.NEWFILESET to be included in the list of filesets generated by the *match_target* option to swinstall if OLDPROD.OLDFILESET has been installed.

Example of supersedes

```
product
    tag NEWPROD
fileset
    tag NEWFILESET
    supersedes OLDPROD.OLDFILESET,r=10.0
```

This designates that installing NEWPROD.NEWFILESET will result in removing the catalog information for OLDPROD.OLDFILESET.

Patch Packaging

If the patch uses the same product tag and fileset tag as the fileset it is patching, then no ancestor is necessarily needed. But, unless this patch patches all previous releases of the base fileset, an ancestor must be specified. If the patch has a different product and fileset tag than the fileset it patches it must also have an ancestor.

A unique product revision must be specified for a patch fileset if that patch has the same product tag and fileset tag as the fileset it is patching.

If this is the first patch of any particular patch stream, then it does not need a *supersedes* attribute. This is true for all original point patches. If this patch replaces one or more other patches, then it

must specify the appropriate *supersedes* attribute.

All patch software objects with the *is_patch* attribute automatically have the built-in patch category included in the list of category_tags.

A fileset that has the *is_patch* attribute will not update a fileset with the same tag as is done with normal filesets. In the case of filesets with *is_patch* set to `true`, the ''revision'' is now a *version distinguishing attribute* at the fileset level.

The *category_tag* and *is_patch* attributes at all other levels of software objects besides fileset are for display and selection purposes only. These attributes are not version distinguishing attributes.

The 1387.2 developers discussed whether patch verification capabilities could be defined that might help the quality of patches, but concluded that this would be outside the scope of their task.

Library/Kernel Patching

In order to patch an archive library, a new file type designed for archive library maintenance is supported. The "file" specification for a .o file also includes the library in which the file is to be placed (using ar).

Example:

```
file -t a  -a /usr/lib/foolib.a  /build/newfile.o \
    /usr/lib/newfile.o

file
    type a
    archive_path /usr/lib/foolib.a
    source /build/newfile.o
    path /usr/lib/newfile.o
```

B.5.3 Serial Format and Multiple Media

The 1387.2 developers discussed the desire for system vendors to have both a boot partition and the serial archive distribution containing the operating system software on a single medium. Numerous strategies were discussed, but no solution could be found that met all the disparate boot strategies. Vendors that have this need may need to create nonconforming media that have both the boot partition and the distribution on the media. They should then copy that distribution into a conforming form.

Separate archives on each medium allows the *swinstall* utility to only request the needed medium (via the *media_sequence_number*), without having to scan each one sequentially. The rule concerning a fileset being in one archive is needed so the *swinstall* utility knows whether to request the next medium when it reaches the end of the current archive.

When packaging a product that contains kernel or prerequisite filesets onto a serial media, *swpackage* should put the filesets on the serial media in the same order in which they will be requested by *swinstall* in order to minimize media rewinds. If the packager and install utilities used are from different implementations, then the amount of rewinds dramatically increases. One possible ordering convention is as follows:

1. Order all products alphabetically by their product tag (and as they are encountered for multiple versions of the same product). Order all filesets alphabetically within each product. This results in the initial ordering for all filesets.

2. Traverse the list front to back, moving all kernel filesets to the front of the list in their same relative order to each other.
3. For each fileset, determine all of the prerequisite dependents for that fileset, meaning filesets that have a prerequisite on this fileset, or a prerequisite on an object containing this fileset. Traverse the list back to front, moving each fileset in front of the first of its dependents in the current list order, considering dependencies across products and chaining of prerequisites as follows:
   - If the first dependent is in the same product, move the fileset just in front of that dependent.
   - If it is in a different product, move it in just in front of all filesets in product that contains the dependent.
   - If a fileset A depends on B which depends on C, move the filesets in C in front of A.
   - Leave the fileset where it is if there are no dependents.
   - Do not move any fileset more than once. Note that circular prerequisites will cause at most two moves, possibly resulting in the original order.
4. Traverse the list front to back, determining a new product order as each new product is encountered. The filesets within each product are also ordered by traversing front to back.
5. Using this product order, place all product catalog files on the serial media; then place all kernel filesets from each product on the serial media (including the prerequisites of kernel filesets); then place all non-kernel filesets from each product on the media.

When installing filesets, use this same order to load kernel filesets before non-kernel filesets, and to load prerequisites before their dependents.

Additional files besides just those for the distribution may be needed on distributions. For example, a software vendor may want to include a copy of the swinstall utility on the media.

Glossary

General Terms

application

The term *application* includes executable programs that use implementations of this Software Administration specification. Such executable programs can include control files.

can

An indication of a permissible optional feature or behavior The implementation must support such features or behaviors as mandatory requirements.

implementation

An object providing the services defined by this Software Administration specification. The word *implementation* is to be interpreted to mean that object, after it has been modified in accordance with the manufacturer's instructions to configure it for conformance with this Software Administration specification, or to select some of the various optional facilities described by this Software Administration specification through customization by local system administrators or operators.

An exception to this meaning occurs when using the term *implementation defined* (see below).

implementation defined

An indication that the implementation provider must define (and document) the requirements for correct program constructs and correct data of a value or behavior.

When the value or behavior in the implementation is designed to be variable or customizable on each instantiation of the system, the implementation provider must document the nature and permissible ranges of this variation in their submission claiming conformance.

may

An indication of an optional feature or behavior of the implementation that is not required by this Software Administration specification, although there is no prohibition against providing it.

A fully conformant implementation (see Section 1.3.1 on page 4 is permitted to use such features, but must not rely on the implementation's actions in such cases. To avoid ambiguity, the reverse sense of *may* is not expressed as *may not*, but as *need not*.

must

A mandatory requirement on the implementation if it is to be fully conformant.

obsolescent

An indication that a certain feature may be considered for withdrawal in future revisions of this Software Administration specification.

should

With respect to implementations, an indication of an implementation recommendation, but not a requirement.

Technical Terms

analysis phase
The steps a software administration utility performs, before modifying the target, while attempting to ensure that the execution of operations on the target will succeed.

API
Application Programming Interface

attribute
A component of an object, possessing a name and one or more values.

autoselect
The automatic selection, within a utility, of software beyond that directly specified by the user in order to meet the dependencies of the user-specified software.

autorecovery
The process of restoring installed software to the state it was in prior to the invocation, and subsequent failure during execution, of the `swinstall` utility.

bundle
A software object used to build groupings from a grouping of other software objects, such as all or parts of other bundles and products. See Section 2.10 on page 20.

catalog
The metadata describing all the software objects that are a part of a single software_collection.

class
Used to describe the structure and attributes of each level of the software hierarchy that is used to organize and manage software files.

client role
Where the software is actually executed or used (as opposed to the target, where it is actually installed). The configuration of software is performed by this role.

command line interface (CLI)
A means of invoking utilities by issuing commands from within a POSIX.2 shell, implying that neither graphics nor windows are required.

common class
Used to define those aspects of different software objects that are the same. The common classes for this Software Administration specification are software_collections, software and software_files. The names of these classes are also used to generically describe any object that shares that common class.

compressed file
a file that has been transformed in a manner intended to reduce its size without loss of information.

containment
A relationship between two objects such that one is said to belong to, or form part of, the other. All objects except software_collection objects must be contained within exactly one object. The containment of software_collection objects is undefined within this Software Administration specification.

control directory
The directory below which the control_files for filesets and products are stored within exported catalogs for installed software.

control_files
The control scripts executed by the utilities, the INFO file describing the files in a fileset, and other files associated with a software object.

control script
A control_file associated with a software object that is executed by the software administration utilities.

corequisite
The specification in a software object that another software object must be installed in conjunction with the installation of the first and configured in conjunction with the configuration of the first.

CRC
Cyclic Redundancy Check

decimal character string
A sequence of characters from the set of decimal digits the first of which must not be the digit zero.

default option
The value for an extended option as defined in a defaults file. See below.

defaults file
A system specific or user specific file that contains the default values for extended options used by the software administration utilities.

dependency
A software object that is a prerequisite, corequisite or exrequisite for a software object.

dependent
A software object which specifies a prerequisite, corequisite or exrequisite on another software object.

developer role
Where software is developed, tested, and maintained.

directory medium
A medium which contains a distribution in a POSIX.1 hierarchical file system format. An example of this is a distribution contained in a POSIX.1 file system format on a CD-ROM.

distribution
A software_collection containing software in the software packaging layout.

distribution catalog
The catalog of metadata for a distribution software_collection. Unlike a catalog for an installed_software object, a distribution catalog is stored in a particular exported catalog structure that is part of the software packaging layout.

distribution path
The pathname below which the catalog describing the distribution is located. If the distribution is on a single medium, all software for it is located below this path.

downdate
Installing an older revision of software than one that is currently installed, into the same location. This is also referred to as downgrading or reverting.

event
An occurrence which may require reporting by the utilities defined in this Software Administration specification. to describe a significant occurrence. The reporting of an event

may cause data to be written to stdout, to stderr, or to a log file.

execution phase
The operations a software administration utility performs that modify the target.

exported catalog
Refers to information organized in the exported catalog structure of the standard packaging layout. It is used for distribution catalogs as well as exporting installed software catalogs using **swlist**.

exrequisite
The specification in a software object that it must not be installed if one or more specific software objects are installed.

extended option
The options that can be specified with the *x* option. These options may be defined in defaults files or options files.

filename
A POSIX.1 filename with characters drawn from the POSIX.1 portable filename character set (see POSIX.1).

filename character string
A sequence of characters from the portable filename character set (see POSIX.1 below), not including the / (slash) character.

fileset
Defines the files that make up a software object, and is the lowest level of software object that can be specified as input to the software administration utilities.

file storage structure
The storage directories in the software packaging layout under which the actual software files for each fileset are located.

fully qualified software_spec
A software_spec which always identifies a software object unambiguously.

graphical user interface (GUI)
A means of presenting function to a user through the use of graphics.

host
A machine that contains software managed by this Software Administration specification.[18]

INDEX file
an exported catalog containing the metadata describing the software objects and attributes for all bundles, products, subproducts and filesets.

INFO file
For each product and fileset, the file within an exported catalog containing the metadata describing the software_file objects and attributes.

---

18. A host may contain both installed_software and distribution software_collections. The name of the host is the starting point for finding all software on that machine managed by this Software Administration specification. The *path* attribute of a software_collection, along with the specification of a host, can be used on the command line to identify a particular software_collection to be managed by this Software Administration specification.

inheritance
The way in which the attribute definitions of a common object class are used as a part of the definition of other object classes. The definition of the new object class includes the definition of the common class plus the additional definitions specific to the new object class.

installed software
Any software object created by the use of the **swinstall** utility.

installed_software
A software_collection containing installed software. This software is in a state ready for use, or ready to be shared by client systems. A directory path on a system and an installed_software catalog together identify a unique installed_software object.

installed_software catalog
The catalog of metadata for an installed_software software_collection. Unlike a catalog for a distribution object, the storage and format of an installed_software catalog is undefined within this Software Administration specification. The ability to dump and restore all or part of an installed_software catalog into an exported catalog structure is included in this Software Administration specification.

installed_software path
The root directory of an installed_software object. The pathname below which all software for that object must be installed.

integer character string
One of a decimal character string, an octal character string, or a hexadecimal character string.

interactive
The behavior of a utility or control_script which requires input from the user during its execution.

kernel fileset
A fileset in which one or more of the referenced files forms part of the kernel, and denoted by having the value of its *is_kernel* attribute set to *true*.

locatable fileset
A fileset for which permission is granted to **swinstall** to install the files in a different location as specified by the user, and denoted by having the value of its *is_locatable* attribute set to *true*.

locatable software
Software that contains locatable filesets.

manager role
Where each task is initiated. The *manager role* is concerned with taking appropriate action at the completion or failure of a task.

metadata
The information kept about software. It consists of the values of the various attributes of each of the objects.

newline string
A white space string (see below), consisting only of the `<newline>` character, which is defined in POSIX.2.

object
An instance in the software hierarchy that can be operated on using the software administration utilities.

octal character string
A sequence of characters from the set of octal digits, the first of which must be the digit zero.

options file
A file that can be specified with the *x* option. This file contains extended option definitions that override default definitions. See also *defaults file*

patch
A fix to existing product that does not provide significant new functionality and is to be installed over an existing installation. This is also refered to as fixing. See also *update*.

packager role
Where software that has been developed is organized in a form suitable for distribution.

pathname
A POSIX.1 pathname with characters drawn from the POSIX.1 portable character set.

pathname character string
A sequence of characters from the portable filename character set as defined in POSIX.1.

portable character string
A sequence of characters from the portable character set as defined in POSIX.2.

prerequisite
The specification in a software object that it must not be installed until after some other software object is installed, and configured until after the other software object is configured,

The manner of honoring such a prerequisite is described in *swinstall* on page 92.

product
A software object used to define a set of related software. Filesets are contained within products.

product specification file (PSF)
The input file used to define the structure and attributes of software objects and related files to be packaged by the **swpackage** utility.

proxy install
A proxy install uses an alternate root directory as the target path.

recovery
The ability of the **swinstall** utility, for a failed software install, to return the system to the state that it was in before the failure, including restoring the files.

reboot fileset
A fileset which, if installed, requires reboot of the operating system to complete its installation, and denoted by having the value of its *is_reboot* attribute set to *true*.

rebooting
An implementation defined procedure generally used to terminate and then restart operations on the target system.

role
The context in which an operation is executed. The utilities in this Software Administration

specification require the ability to perform operations on more than one system, perhaps by more than one person. These operations are separated into distinct roles developer, packager, manager, source, target, and client.

selection phase
The set of steps performed by software administration utility to process selections and options.

serial medium
A medium which contains a POSIX.1 extended **tar** or extended **cpio** archive. See POSIX.1.

session
An execution of a software administration command from initiation to completion on all applicable roles.

shell token string
A sequence of shell tokens. Shell tokens are defined in POSIX.2.

software
A generic term referring to software objects or a structured set of files. This term can refer to the objects forming the hierarchical structure (software objects), or to the actual files and control_files (software files).

software_collection
A grouping of software objects that are managed by the software administration utilities. Software_collections are the sources and targets of these utilities. This Software Administration specification defines two types of software_collections installed_software and distributions.

software common class
The common class describing the common attributes associated with the hierarchical structure of software objects defined by this Software Administration specification.

software definition files
The files containing the software structure and detailed attributes for distributions, installed_software, bundles, products, subproducts, filesets, files, and control_files.

To communicate metadata information relating to both distributions and installed software, software definition files serve as input to, or output from, the various software administration utilities. The format used by software administration utilities to store metadata relating to installed software is undefined.

software file
A generic term referring to the files and control_files that are contained within software objects and managed by the utilities in this Software Administration specification.

software_file common class
The common class that relates the two types of files defined by this Software Administration specification, namely the actual files that make up the software, plus the control_files that are executed by the utilities when operating on software.

software_files
A generic term referring to file and control_file objects (those that share the same software_file common class).

software hierarchy
Hierarchical organization of objects that are managed by the software administration utilities.

software location

The directory relative to the installed_software root directory where the relocatable files of the software have been located.

software object

An object which inherits attributes of the software common class, meaning a bundle, product, subproduct or fileset object.

software packaging layout

The format for software in a distribution. It contains the metadata for the distribution catalog in a well defined exported form, as well as the files for the software objects in that distribution.

software pattern match string

A sequence of one or more strings, each made up of a sequence of one or more characters from the shell ''Pattern Matching Notation'' strings described in POSIX.2. If there are two or more strings, the strings are separated by the | character. The match is true if any of the sequences of strings match according to POSIX.2. A software pattern match string must be portable character string.

software_spec

A string that is used to identify one or more software objects for input to a software administration utility.

source

The specification of a source distribution object for a software administration utility. The source host provides a means to locate the source role and the source path is a path accessible to the source host.

source host

The host portion of a source specification.

source path

The pathname portion of a source specification.

source role

Where the software exists in a form suitable for distribution, forming a context for the establishment of a repository of software from which the manager may choose to distribute to targets. Software exists in the source until it is removed by a task initiated by the manager. The source role provides a repository where software may be stored, and provides access for those roles that require the software.

subproduct

A software object which is a grouping of software filesets and other subproducts within a product.

symbolic link (symlink)

A type of file that contains a pathname. Rather than containing data itself, this type of file resolves to another, as defined by the contained pathname. The way in which this type of file is handled by implementations of this Software Administration specification is undefined.[19]

---

19. It is not the intention of this Software Administration specification to define symbolic links in a manner inconsistent with POSIX.1. However, no approved POSIX standard currently contains symbolic links. This definition is a placeholder until such time as an approved standard provides the definition.

system

An implementation of this Software Administration specification.

target

The specification of a target distribution object, or installed software object, for a software administration utility. The target host provides a means to locate the target role and the target path is a path accessible to the target host.

target host

The host portion of a target specification.

target path

The pathname portion of a target specification.

target role

Where software is installed, removed, listed, and otherwise operated on by the utilities. For example, when installing software, the target is where software is installed after having been delivered from a source. As another example, the target for a copy operation command refers to the distribution to which products are added. For management operations like removing software, the target refers to either the installed_software objects or the distributions from which software is being removed.

update

Installing a newer revision of software than one that is currently installed, into the same location. This is also referred to as upgrading. This implies that two filesets that have the same tag and same product tags can not be installed in, or exist in, the same location at the same time (with the exception of patching enhancements). Updates (or *upgrades*) generally involve software releases providing significant new functionality that can be installed for the first time or can update an existing installation. Patches (or *fixes*) generally involve fixes to existing products that do not provide significant new functionality and are to be installed over an existing installation. Updates generally involve software releases providing significant new functionality. See also *patch*.

UTF-8

UCS Transformation Format 8, as defined in ISO/IEC 10646-1: 1993, See referenced documents.

vendor

An item, such as a nonstandard attribute, that is defined by the vendor that created (packaged) the software.

vendor-supplied

An item, such as a control file, that is supplied by the creator (packager) of the software.

version

A unique identification of software based on the attributes of the software. Version differentiates software objects with the same value of the *tag* attribute. Versions of bundles or products have the same value of the *tag* attribute and will differ by the value of at least one of *revision*, *architecture*, *vendor_tag*, *location* or *qualifier* attributes. The *location* and *qualifier* attributes only apply to software in installed_software software_collections. A fileset is considered a version of another fileset if they have the same *fileset* tag and their respective products have the same *product* tag.

white space string

A sequence of one or more white space characters (as defined in POSIX.2.

wildcard character

One of `* ? [` (asterisk, question mark, open bracket). Such characters are used in software pattern match strings

Index

machine 175
release 176
size_t 179
st_gid 32
st_gid 32
sysname 176
version 176
alternate root directory 18, 36, 43, 96, 181, 187, 196-197, 214, 216
analysis phase 244
API 244
application 243
attribute 244
- vendor defined 135

attributes
- category 13
- dfiles 126
- names as keywords 5
- of bundle 20
- of control file 33
- of distribution 9
- of file 31
- of fileset 24
- of installed software 11
- of media 10
- of product 16
- of software 14
- of software collection 7
- of software file 29
- of subproduct 28
- of vendor 12
- pfiles 127
- semantics of 134
- uname 42, 122, 182
- version distinguishing 9, 11, 16, 20, 92, 104

attributesqq, version distinguishing 179, 222
autorecovery 244
autoselect 244
bundle 244
can 243
catalog 7, 244
class 244
client role 244
command line interface (CLI) 244
command_line_options, definition 193
common class 244
compressed file 244
conformance 4
conforming distribution 33
conforming implementation 4
containment 244
control directory 244
control script 245
control script example 158
control_files 245
corequisite 86, 183, 245
CRC 245
decimal character string 245
default option 245
defaults file 245
defaults file example 149
dependency 245
dependency_spec definition 39
dependent 245
developer role 245
devices 237
devmajor 32
devminor 32
directory medium 245
distributed interoperability 161
distribution 245
distribution catalog 245
distribution path 245
distribution, bootable 170
distribution, compressed 170
downdate 245
event 245
execution phase 246
existing practice 167
exported catalog 7, 246
exrequisite 86, 246
extended option 246
file loading 96
file storage structure 246
filename 246
filename character string 246
fileset 246
fully qualified software_spec 246
graphical user interface (GUI) 246
hard link 185
host 246

implementation....243
implementation defined....33, 47, 243
INDEX file....246
INDEX file example....155
INFO file....246
INFO file example....157
inheritance....247
installed software....247
installed_software....247
installed_software catalog....247
installed_software path....247
integer character string....247
interactive....247
interoperability....161
kernel....67
kernel fileset....247
keyword...5, 53-54, 130, 133-135, 144, 221-222, 236
keywords
  all_filesets....138
  applied_patches....139
  architecture....137-138
  archive_path....141
  catalog....136
  category_tag....137-139
  checkinstall....142
  checkremove....142
  cksum....140-141
  compressed_cksum....140-141
  compressed_size....140-141
  compression_state....140-141
  compression_type....140-141
  configure....142
  contents....137
  control_directory....138-139
  control_file....142
  copyright....137-138
  corequisites....139
  create_time....137-139
  description....136-139
  dfiles....136
  directory....137-138, 143
  exrequisites....139
  file....143-144
  files....135
  filesets....139
  file_permissions....146
  fix....142
  gid....141
  group....141
  instance_id....137-138
  interpreter....140
  is_kernel....139
  is_locatable....137-139
  is_patch....137-139
  is_reboot....139
  is_volatile....141
  layout_version....135-136
  link_source....141
  location....137-139
  machine_type....137-138
  major....141
  media_sequence_number....139
  minor....141
  mode....141
  mod_time....137-139
  mtime....141
  names of attributes....5
  number....137-138
  os_name....137-138
  os_release....137-138
  os_version....137-138
  owner....141
  patch_state....139
  path....135-136, 140-141, 146
  pfiles....135-136
  postinstall....142
  postkernel_type....138
  postremove....142
  preinstall....142
  preremove....142
  prerequisites....139
  qualifier....137-138
  request....142
  result....140
  revision....136-139, 141
  semantics of....134
  sequence_number....135
  size....137-141
  source....140-141
  space....142
  state....139
  subproducts....139
  superseded_by....139
  supersedes....139
  tag....136-140
  title....136-139
  type....141
  uid....141
  unconfigure....142
  unpostinstall....142
  unpreinstall....142
  vendor defined....135

vendor_tag 137-138
verify 142
keywords, distribution 236
limited conformance 4
locatable fileset 247
locatable software 247
manager role 247
may 243
media 147-148
media sequence number 67, 140
medium 147-148
metadata 247
must 243
newline string 247
object 248
obsolescent 243
octal character string 248
options file 248
options_file, definition 194
packager role 248
patch 248
pathname 248
pathname character string 248
portable character string 248
POSIX.1 3
POSIX.2 3
prerequisite 86, 95, 99, 113, 147, 166, 183
.... 197-199, 240-241, 248
product 248
product specification file (PSF) 248
proxy install 56, 164, 181, 187, 214, 216, 248
PSF example 152
pw_name 32
reboot fileset 248
rebooting 248
recovery 248
role 248
sample code
control script 158
defaults file 149
INDEX file 155
INFO file 157
PSF 152
software packaging layout 154
selection phase 249
serial medium 249
session 249
shell token string 249
should 243
software 249
software common class 249
software definition files 249
software file 249
software hierarchy 249
software location 250
software object 250
software packaging layout 250
software packaging layout example 154
software pattern match string 250
software_collection 249
software_collection_spec definition 42
software_definition_file definition 130
software_file common class 249
software_files 249
software_option definition 52
software_selections definition 38
software_spec 250
software_spec definition 38
source 250
source host 250
source path 250
source role 250
source, well-known 201
state, restoring 101
states
available 201
configured 201
corrupt 201
installed 201
removed 201
transient 201
st_mode 32
st_mtime 32
st_size 30
st_uid 32
subproduct 250
SVR4 191
swask **78**, 204
swconfig **81**, 207
swcopy **85**, 210
swinstall **90**, 212
swlist **104**, 221
swmodify **108**, 226
swpackage **111**, 228
swremove **115**, 230
swverify **121**, 233
symbolic link 185
symbolic link (symlink) 250
sysadmin_option, definition 192
system 251
tar 32
target 251

target host 251
target path..251
target role 251
uname....................................18, 21-22, 42, 122, 135
update 251
UTF-8...251
vendor...251
vendor-supplied 251
version 251
version distinguishing....................................92, 104
white space string..251
wildcard character...252