```
**********************************************************
   24172 Wed Sep 16 00:43:13 2015
new/usr/src/cmd/svc/svccfg/svccfg_engine.c
1565 svccfg cleanup needs to suck it up and finish the job
**********************************************************
______unchanged_portion_omitted_

 825 /*
 826  * Walk each service and get its manifest file.
 827  *
 828  * If the file exists check instance support, and cleanup any
 829  * stale instances.
 830  *
 831  * If the file doesn't exist tear down the service and/or instances
 832  * that are no longer supported by files.
 833  */
 834 int
 835 engine_cleanup(int flags)
 836 {
 837         boolean_t               activity = B_TRUE;
 838         int                     dont_exit;
 839 #endif /* ! codereview */
 840         int                     r = -1;

 842         lscf_prep_hndl();

 844         if (flags == 1) {
 845                 activity = B_FALSE;
 846         }

 848         dont_exit = est->sc_cmd_flags & SC_CMD_DONT_EXIT;
 849         est->sc_cmd_flags |= SC_CMD_DONT_EXIT;

 851 #endif /* ! codereview */
 852         if (scf_walk_fmri(g_hndl, 0, NULL, SCF_WALK_SERVICE|SCF_WALK_NOINSTANCE,
 853             lscf_service_cleanup, (void *)activity, NULL,
 854             uu_warn) == SCF_SUCCESS)
 855                 r = 0;

 857         if (dont_exit == 0)
 858                 est->sc_cmd_flags &= ~SC_CMD_DONT_EXIT;
 859 #endif /* ! codereview */

 861         (void) lscf_hash_cleanup();

 863         return (r);
 864 }

 866 static int
 867 apply_profile(manifest_info_t *info, int apply_changes)
 868 {
 869         bundle_t *b = internal_bundle_new();

 871         if (lxml_get_bundle_file(b, info->mi_path, SVCCFG_OP_APPLY) != 0) {
 872                 internal_bundle_free(b);
 873                 return (-1);
 874         }

 876         if (!apply_changes) {   /* we don't want to apply, just test */
 877                 internal_bundle_free(b);
 878                 return (0);
 879         }

 881         if (lscf_bundle_apply(b, info->mi_path) != 0) {
 882                 internal_bundle_free(b);
 883                 return (-1);
```



```
 884         }

 886         internal_bundle_free(b);

 888         if (info->mi_prop) {
 889                 apply_action_t apply;
 890                 char *errstr;

 892                 apply = (est->sc_in_emi == 1) ? APPLY_LATE : APPLY_NONE;
 893                 if (mhash_store_entry(g_hndl, info->mi_prop, info->mi_path,
 894                     info->mi_hash, apply, &errstr)) {
 895                         semerr(errstr);
 896                 }
 897         }

 899         return (0);
 900 }

 902 int
 903 engine_apply(const char *file, int apply_changes)
 904 {
 905         int rc = 0;
 906         int isdir;
 907         int dont_exit;
 908         int profile_count;
 909         int fm_flags;
 910         struct stat sb;
 911         manifest_info_t **profiles = NULL;
 912         manifest_info_t **entry;
 913         manifest_info_t *pfile;

 915         lscf_prep_hndl();

 917         /* Determine which profile(s) must be applied. */

 919         profile_count = 0;
 920         fm_flags = BUNDLE_PROF | CHECKHASH;

 922         /* Determine if argument is a directory or file. */
 923         if (stat(file, &sb) == -1) {
 924                 semerr(gettext("Unable to stat file %s.  %s\n"), file,
 925                     strerror(errno));
 926                 rc = -1;
 927                 goto out;
 928         }

 930         if (sb.st_mode & S_IFDIR) {
 931                 fm_flags |= CHECKEXT;
 932                 isdir = 1;
 933         } else if (sb.st_mode & S_IFREG) {
 934                 isdir = 0;
 935         } else {
 936                 semerr(gettext("%s is not a directory or regular "
 937                     "file\n"), file);
 938                 rc = -1;
 939                 goto out;
 940         }

 942         /* Get list of profiles to be applied. */
 943         if ((profile_count = find_manifests(g_hndl, file, &profiles,
 944             fm_flags)) < 0) {

 946                 if (isdir) {
 947                         semerr(gettext("Could not hash directory %s\n"), file);
 948                 } else {
 949                         semerr(gettext("Could not hash file %s\n"), file);
```

```
950                 }
951                 rc = -1;
952                 goto out;
953         }

955         if (profile_count == 0) {
956                 /* No profiles to process. */
957                 if (g_verbose) {
958                         warn(gettext("No changes were necessary\n"));
959                 }
960                 goto out;
961         }

963         /*
964          * We don't want to exit if we encounter an error.  We should go ahead
965          * and process all of the profiles.
966          */
967         dont_exit = est->sc_cmd_flags & SC_CMD_DONT_EXIT;
968         est->sc_cmd_flags |= SC_CMD_DONT_EXIT;

970         for (entry = profiles; *entry != NULL; entry++) {
971                 pfile = *entry;

973                 if (apply_profile(pfile, apply_changes) == 0) {
974                         if (g_verbose) {
975                                 warn(gettext("Successfully applied: %s\n"),
976                                     pfile->mi_path);
977                         }
978                 } else {
979                         warn(gettext("WARNING: Failed to apply %s\n"),
980                             pfile->mi_path);
981                         rc = -1;
982                 }
983         }

985         if (dont_exit == 0)
986                 est->sc_cmd_flags &= ~SC_CMD_DONT_EXIT;

988         /* exit(1) appropriately if any profile failed to be applied. */
989         if ((rc == -1) &&
990             (est->sc_cmd_flags & (SC_CMD_IACTIVE | SC_CMD_DONT_EXIT)) == 0) {
991                 free_manifest_array(profiles);
992                 exit(1);
993         }

995 out:
996         free_manifest_array(profiles);
997         return (rc);
998 }

1000 int
1001 engine_restore(const char *file)
1002 {
1003         bundle_t *b;

1005         lscf_prep_hndl();

1007         b = internal_bundle_new();

1009         if (lxml_get_bundle_file(b, file, SVCCFG_OP_RESTORE) != 0) {
1010                 internal_bundle_free(b);
1011                 return (-1);
1012         }

1014         if (lscf_bundle_import(b, file, SCI_NOSNAP) != 0) {
1015                 internal_bundle_free(b);
```



```
1016                 return (-1);
1017         }

1019         internal_bundle_free(b);

1021         return (0);
1022 }

1024 int
1025 engine_set(uu_list_t *args)
1026 {
1027         uu_list_walk_t *walk;
1028         string_list_t *slp;

1030         if (uu_list_first(args) == NULL) {
1031                 /* Display current options. */
1032                 if (!g_verbose)
1033                         (void) fputs("no", stdout);
1034                 (void) puts("verbose");

1036                 return (0);
1037         }

1039         walk = uu_list_walk_start(args, UU_DEFAULT);
1040         if (walk == NULL)
1041                 uu_die(gettext("Couldn't read arguments"));

1043         /* Use getopt? */
1044         for (slp = uu_list_walk_next(walk);
1045             slp != NULL;
1046             slp = uu_list_walk_next(walk)) {
1047                 if (slp->str[0] == '-') {
1048                         char *op;

1050                         for (op = &slp->str[1]; *op != '\0'; ++op) {
1051                                 switch (*op) {
1052                                 case 'v':
1053                                         g_verbose = 1;
1054                                         break;

1056                                 case 'V':
1057                                         g_verbose = 0;
1058                                         break;

1060                                 default:
1061                                         warn(gettext("Unknown option -%c.\n"),
1062                                             *op);
1063                                 }
1064                         }
1065                 } else {
1066                         warn(gettext("No non-flag arguments defined.\n"));
1067                 }
1068         }

1070         return (0);
1071 }

1073 void
1074 help(int com)
1075 {
1076         int i;

1078         if (com == 0) {
1079                 warn(gettext("General commands:  help set repository end\n"
1080                     "Manifest commands:  inventory validate import export "
1081                     "archive\n"
```

```
1082                    "Profile commands:   apply extract\n"
1083                    "Entity commands:    list select unselect add delete "
1084                    "describe\n"
1085                    "Snapshot commands:  listsnap selectsnap revert\n"
1086                    "Instance commands:  refresh\n"
1087                    "Property group commands: listpg addpg delpg\n"
1088                    "Property commands:  listprop setprop delprop editprop\n"
1089                    "Property value commands: addpropvalue delpropvalue "
1090                    "setenv unsetenv\n"
1091                    "Notification parameters: "
1092                    "listnotify setnotify delnotify\n"));
1093                return;
1094        }

1096        for (i = 0; help_messages[i].message != NULL; ++i) {
1097                if (help_messages[i].token == com) {
1098                        warn(gettext("Usage: %s\n"),
1099                            gettext(help_messages[i].message));
1100                        return;
1101                }
1102        }

1104        warn(gettext("Unknown command.\n"));
1105 }
```

```
**********************************************************
  101567 Wed Sep 16 00:43:13 2015
new/usr/src/cmd/svc/svccfg/svccfg_xml.c
335 service manifest does not support multiple manpage references for the same k
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2004, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 */

#endif /* ! codereview */

/*
 * XML document manipulation routines
 *
 * These routines provide translation to and from the internal representation to
 * XML.  Directionally-oriented verbs are with respect to the external source,
 * so lxml_get_service() fetches a service from the XML file into the
 * internal representation.
 */

#include <libxml/parser.h>
#include <libxml/xinclude.h>

#include <assert.h>
#include <ctype.h>
#include <errno.h>
#include <libintl.h>
#include <libscf.h>
#include <libscf_priv.h>
#include <libuutil.h>
#include <sasl/saslutil.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#include <sys/param.h>
#include "manifest_hash.h"

#include "svccfg.h"
```



```
#include "notify_params.h"

/*
 * snprintf(3C) format strings for constructing property names that include
 * the locale designation.  Use %s to indicate where the locale should go.
 *
 * The VALUE_* symbols are an exception.  The firs %s will be replaced with
 * "value_".  The second %s will be replaced by the name of the value and
 * %%s will be replaced by the locale designation.  These formats are
 * processed twice by snprintf(3C).  The first time captures the value name
 * and the second time captures the locale.
 */
#define LOCALE_ONLY_FMT         ("%s")
#define COMMON_NAME_FMT         ("common_name_%s")
#define DESCRIPTION_FMT         ("description_%s")
#define UNITS_FMT               ("units_%s")
#define VALUE_COMMON_NAME_FMT   ("%s%s_common_name_%%s")
#define VALUE_DESCRIPTION_FMT   ("%s%s_description_%%s")

/* Attribute names */
const char * const delete_attr = "delete";
const char * const enabled_attr = "enabled";
const char * const lang_attr = "lang";
const char * const manpath_attr = "manpath";
const char * const max_attr = "max";
const char * const min_attr = "min";
const char * const name_attr = "name";
const char * const override_attr = "override";
const char * const required_attr = "required";
const char * const section_attr = "section";
const char * const set_attr = "set";
const char * const target_attr = "target";
const char * const timeout_seconds_attr = "timeout_seconds";
const char * const title_attr = "title";
const char * const type_attr = "type";
const char * const uri_attr = "uri";
const char * const value_attr = "value";
const char * const version_attr = "version";
const char * const xml_lang_attr = "xml:lang";
const char * const active_attr = "active";

/* Attribute values */
const char * const all_value = "all";

const char * const true = "true";
const char * const false = "false";

/*
 * The following list must be kept in the same order as that of
 * element_t array
 */
static const char *lxml_elements[] = {
        "astring_list",                 /* SC_ASTRING */
        "boolean_list",                 /* SC_BOOLEAN */
        "cardinality",                  /* SC_CARDINALITY */
        "choices",                      /* SC_CHOICES */
        "common_name",                  /* SC_COMMON_NAME */
        "constraints",                  /* SC_CONSTRAINTS */
        "count_list",                   /* SC_COUNT */
        "create_default_instance",      /* SC_INSTANCE_CREATE_DEFAULT */
        "dependency",                   /* SC_DEPENDENCY */
        "dependent",                    /* SC_DEPENDENT */
        "description",                  /* SC_DESCRIPTION */
        "doc_link",                     /* SC_DOC_LINK */
        "documentation",                /* SC_DOCUMENTATION */
        "enabled",                      /* SC_ENABLED */
```

```
128         "event",                        /* SC_EVENT */
129         "exec_method",                  /* SC_EXEC_METHOD */
130         "fmri_list",                    /* SC_FMRI */
131         "host_list",                    /* SC_HOST */
132         "hostname_list",                /* SC_HOSTNAME */
133         "include_values",               /* SC_INCLUDE_VALUES */
134         "instance",                     /* SC_INSTANCE */
135         "integer_list",                 /* SC_INTEGER */
136         "internal_separators",          /* SC_INTERNAL_SEPARATORS */
137         "loctext",                      /* SC_LOCTEXT */
138         "manpage",                      /* SC_MANPAGE */
139         "method_context",               /* SC_METHOD_CONTEXT */
140         "method_credential",            /* SC_METHOD_CREDENTIAL */
141         "method_profile",               /* SC_METHOD_PROFILE */
142         "method_environment",           /* SC_METHOD_ENVIRONMENT */
143         "envvar",                       /* SC_METHOD_ENVVAR */
144         "net_address_list",             /* SC_NET_ADDR */
145         "net_address_v4_list",          /* SC_NET_ADDR_V4 */
146         "net_address_v6_list",          /* SC_NET_ADDR_V6 */
147         "notification_parameters",      /* SC_NOTIFICATION_PARAMETERS */
148         "opaque_list",                  /* SC_OPAQUE */
149         "parameter",                    /* SC_PARAMETER */
150         "paramval",                     /* SC_PARAMVAL */
151         "pg_pattern",                   /* SC_PG_PATTERN */
152         "prop_pattern",                 /* SC_PROP_PATTERN */
153         "property",                     /* SC_PROPERTY */
154         "property_group",               /* SC_PROPERTY_GROUP */
155         "propval",                      /* SC_PROPVAL */
156         "range",                        /* SC_RANGE */
157         "restarter",                    /* SC_RESTARTER */
158         "service",                      /* SC_SERVICE */
159         "service_bundle",               /* SC_SERVICE_BUNDLE */
160         "service_fmri",                 /* SC_SERVICE_FMRI */
161         "single_instance",              /* SC_INSTANCE_SINGLE */
162         "stability",                    /* SC_STABILITY */
163         "template",                     /* SC_TEMPLATE */
164         "time_list",                    /* SC_TIME */
165         "type",                         /* SC_TYPE */
166         "units",                        /* SC_UNITS */
167         "uri_list",                     /* SC_URI */
168         "ustring_list",                 /* SC_USTRING */
169         "value",                        /* SC_VALUE */
170         "value_node",                   /* SC_VALUE_NODE */
171         "values",                       /* SC_VALUES */
172         "visibility",                   /* SC_VISIBILITY */
173         "xi:fallback",                  /* SC_XI_FALLBACK */
174         "xi:include"                    /* SC_XI_INCLUDE */
175 };

177 /*
178  * The following list must be kept in the same order as that of
179  * element_t array
180  */
181 static const char *lxml_prop_types[] = {
182         "astring",                      /* SC_ASTRING */
183         "boolean",                      /* SC_BOOLEAN */
184         "",                             /* SC_CARDINALITY */
185         "",                             /* SC_CHOICES */
186         "",                             /* SC_COMMON_NAME */
187         "",                             /* SC_CONSTRAINTS */
188         "count",                        /* SC_COUNT */
189         "",                             /* SC_INSTANCE_CREATE_DEFAULT */
190         "",                             /* SC_DEPENDENCY */
191         "",                             /* SC_DEPENDENT */
192         "",                             /* SC_DESCRIPTION */
193         "",                             /* SC_DOC_LINK */
```



```
194         "",                             /* SC_DOCUMENTATION */
195         "",                             /* SC_ENABLED */
196         "",                             /* SC_EVENT */
197         "",                             /* SC_EXEC_METHOD */
198         "fmri",                         /* SC_FMRI */
199         "host",                         /* SC_HOST */
200         "hostname",                     /* SC_HOSTNAME */
201         "",                             /* SC_INCLUDE_VALUES */
202         "",                             /* SC_INSTANCE */
203         "integer",                      /* SC_INTEGER */
204         "",                             /* SC_INTERNAL_SEPARATORS */
205         "",                             /* SC_LOCTEXT */
206         "",                             /* SC_MANPAGE */
207         "",                             /* SC_METHOD_CONTEXT */
208         "",                             /* SC_METHOD_CREDENTIAL */
209         "",                             /* SC_METHOD_PROFILE */
210         "",                             /* SC_METHOD_ENVIRONMENT */
211         "",                             /* SC_METHOD_ENVVAR */
212         "net_address",                  /* SC_NET_ADDR */
213         "net_address_v4",               /* SC_NET_ADDR_V4 */
214         "net_address_v6",               /* SC_NET_ADDR_V6 */
215         "",                             /* SC_NOTIFICATION_PARAMETERS */
216         "opaque",                       /* SC_OPAQUE */
217         "",                             /* SC_PARAMETER */
218         "",                             /* SC_PARAMVAL */
219         "",                             /* SC_PG_PATTERN */
220         "",                             /* SC_PROP_PATTERN */
221         "",                             /* SC_PROPERTY */
222         "",                             /* SC_PROPERTY_GROUP */
223         "",                             /* SC_PROPVAL */
224         "",                             /* SC_RANGE */
225         "",                             /* SC_RESTARTER */
226         "",                             /* SC_SERVICE */
227         "",                             /* SC_SERVICE_BUNDLE */
228         "",                             /* SC_SERVICE_FMRI */
229         "",                             /* SC_INSTANCE_SINGLE */
230         "",                             /* SC_STABILITY */
231         "",                             /* SC_TEMPLATE */
232         "time",                         /* SC_TIME */
233         "",                             /* SC_TYPE */
234         "",                             /* SC_UNITS */
235         "uri",                          /* SC_URI */
236         "ustring",                      /* SC_USTRING */
237         "",                             /* SC_VALUE */
238         "",                             /* SC_VALUE_NODE */
239         "",                             /* SC_VALUES */
240         "",                             /* SC_VISIBILITY */
241         "",                             /* SC_XI_FALLBACK */
242         ""                              /* SC_XI_INCLUDE */
243 };

245 int
246 lxml_init()
247 {
248         if (getenv("SVCCFG_NOVALIDATE") == NULL) {
249                 /*
250                  * DTD validation, with line numbers.
251                  */
252                 (void) xmlLineNumbersDefault(1);
253                 xmlLoadExtDtdDefaultValue |= XML_DETECT_IDS;
254                 xmlLoadExtDtdDefaultValue |= XML_COMPLETE_ATTRS;
255         }

257         return (0);
258 }
```

```
260 static bundle_type_t
261 lxml_xlate_bundle_type(xmlChar *type)
262 {
263         if (xmlStrcmp(type, (const xmlChar *)"manifest") == 0)
264                 return (SVCCFG_MANIFEST);
265
266         if (xmlStrcmp(type, (const xmlChar *)"profile") == 0)
267                 return (SVCCFG_PROFILE);
268
269         if (xmlStrcmp(type, (const xmlChar *)"archive") == 0)
270                 return (SVCCFG_ARCHIVE);
271
272         return (SVCCFG_UNKNOWN_BUNDLE);
273 }
274
275 static service_type_t
276 lxml_xlate_service_type(xmlChar *type)
277 {
278         if (xmlStrcmp(type, (const xmlChar *)"service") == 0)
279                 return (SVCCFG_SERVICE);
280
281         if (xmlStrcmp(type, (const xmlChar *)"restarter") == 0)
282                 return (SVCCFG_RESTARTER);
283
284         if (xmlStrcmp(type, (const xmlChar *)"milestone") == 0)
285                 return (SVCCFG_MILESTONE);
286
287         return (SVCCFG_UNKNOWN_SERVICE);
288 }
289
290 static element_t
291 lxml_xlate_element(const xmlChar *tag)
292 {
293         int i;
294
295         for (i = 0; i < sizeof (lxml_elements) / sizeof (char *); i++)
296                 if (xmlStrcmp(tag, (const xmlChar *)lxml_elements[i]) == 0)
297                         return ((element_t)i);
298
299         return ((element_t)-1);
300 }
301
302 static uint_t
303 lxml_xlate_boolean(const xmlChar *value)
304 {
305         if (xmlStrcmp(value, (const xmlChar *)true) == 0)
306                 return (1);
307
308         if (xmlStrcmp(value, (const xmlChar *)false) == 0)
309                 return (0);
310
311         uu_die(gettext("illegal boolean value \"%s\"\n"), value);
312
313         /*NOTREACHED*/
314 }
315
316 static scf_type_t
317 lxml_element_to_type(element_t type)
318 {
319         switch (type) {
320         case SC_ASTRING:        return (SCF_TYPE_ASTRING);
321         case SC_BOOLEAN:        return (SCF_TYPE_BOOLEAN);
322         case SC_COUNT:          return (SCF_TYPE_COUNT);
323         case SC_FMRI:           return (SCF_TYPE_FMRI);
324         case SC_HOST:           return (SCF_TYPE_HOST);
325         case SC_HOSTNAME:       return (SCF_TYPE_HOSTNAME);
```



```
326         case SC_INTEGER:        return (SCF_TYPE_INTEGER);
327         case SC_NET_ADDR:       return (SCF_TYPE_NET_ADDR);
328         case SC_NET_ADDR_V4:    return (SCF_TYPE_NET_ADDR_V4);
329         case SC_NET_ADDR_V6:    return (SCF_TYPE_NET_ADDR_V6);
330         case SC_OPAQUE:         return (SCF_TYPE_OPAQUE);
331         case SC_TIME:           return (SCF_TYPE_TIME);
332         case SC_URI:            return (SCF_TYPE_URI);
333         case SC_USTRING:        return (SCF_TYPE_USTRING);
334
335         default:
336                 uu_die(gettext("unknown value type (%d)\n"), type);
337         }
338
339         /* NOTREACHED */
340 }
341
342 static element_t
343 lxml_type_to_element(scf_type_t type)
344 {
345         switch (type) {
346         case SCF_TYPE_ASTRING:          return (SC_ASTRING);
347         case SCF_TYPE_BOOLEAN:          return (SC_BOOLEAN);
348         case SCF_TYPE_COUNT:            return (SC_COUNT);
349         case SCF_TYPE_FMRI:             return (SC_FMRI);
350         case SCF_TYPE_HOST:             return (SC_HOST);
351         case SCF_TYPE_HOSTNAME:         return (SC_HOSTNAME);
352         case SCF_TYPE_INTEGER:          return (SC_INTEGER);
353         case SCF_TYPE_NET_ADDR:         return (SC_NET_ADDR);
354         case SCF_TYPE_NET_ADDR_V4:      return (SC_NET_ADDR_V4);
355         case SCF_TYPE_NET_ADDR_V6:      return (SC_NET_ADDR_V6);
356         case SCF_TYPE_OPAQUE:           return (SC_OPAQUE);
357         case SCF_TYPE_TIME:             return (SC_TIME);
358         case SCF_TYPE_URI:              return (SC_URI);
359         case SCF_TYPE_USTRING:          return (SC_USTRING);
360
361         default:
362                 uu_die(gettext("unknown value type (%d)\n"), type);
363         }
364
365         /* NOTREACHED */
366 }
367
368 /*
369  * Create a SCF_TYPE_BOOLEAN property name pname and attach it to the
370  * property group at pgrp.  The value of the property will be set from the
371  * attribute named attr.  attr must have a value of 0, 1, true or false.
372  *
373  * Zero is returned on success.  An error is indicated by -1.  It indicates
374  * that either the attribute had an invalid value or that we could not
375  * attach the property to pgrp.  The attribute should not have an invalid
376  * value if the DTD is correctly written.
377  */
378 static int
379 new_bool_prop_from_attr(pgroup_t *pgrp, const char *pname, xmlNodePtr n,
380     const char *attr)
381 {
382         uint64_t bool;
383         xmlChar *val;
384         property_t *p;
385         int r;
386
387         val = xmlGetProp(n, (xmlChar *)attr);
388         if (val == NULL)
389                 return (0);
390
391         if ((xmlStrcmp(val, (xmlChar *)"0") == 0) ||
```

```
392             (xmlStrcmp(val, (xmlChar *)"false") == 0)) {
393                 bool = 0;
394         } else if ((xmlStrcmp(val, (xmlChar *)"1") == 0) ||
395             (xmlStrcmp(val, (xmlChar *)"true") == 0)) {
396                 bool = 1;
397         } else {
398                 xmlFree(val);
399                 return (-1);
400         }
401         xmlFree(val);
402         p = internal_property_create(pname, SCF_TYPE_BOOLEAN, 1, bool);
403         r = internal_attach_property(pgrp, p);

405         if (r != 0)
406                 internal_property_free(p);

408         return (r);
409 }

411 static int
412 new_str_prop_from_attr(pgroup_t *pgrp, const char *pname, scf_type_t ty,
413     xmlNodePtr n, const char *attr)
414 {
415         xmlChar *val;
416         property_t *p;
417         int r;

419         val = xmlGetProp(n, (xmlChar *)attr);

421         p = internal_property_create(pname, ty, 1, val);
422         r = internal_attach_property(pgrp, p);

424         if (r != 0)
425                 internal_property_free(p);

427         return (r);
428 }

430 static int
431 new_opt_str_prop_from_attr(pgroup_t *pgrp, const char *pname, scf_type_t ty,
432     xmlNodePtr n, const char *attr, const char *dflt)
433 {
434         xmlChar *val;
435         property_t *p;
436         int r;

438         val = xmlGetProp(n, (xmlChar *)attr);
439         if (val == NULL) {
440                 if (dflt == NULL) {
441                         /*
442                          * A missing attribute is considered to be a
443                          * success in this function, because many of the
444                          * attributes are optional.  Missing non-optional
445                          * attributes will be detected later when template
446                          * validation is done.
447                          */
448                         return (0);
449                 } else {
450                         val = (xmlChar *)dflt;
451                 }
452         }

454         p = internal_property_create(pname, ty, 1, val);
455         r = internal_attach_property(pgrp, p);

457         if (r != 0)
```



```
458                 internal_property_free(p);

460         return (r);
461 }

463 static int
464 lxml_ignorable_block(xmlNodePtr n)
465 {
466         return ((xmlStrcmp(n->name, (xmlChar *)"text") == 0 ||
467             xmlStrcmp(n->name, (xmlChar *)"comment") == 0) ? 1 : 0);
468 }

470 static void
471 lxml_validate_element(xmlNodePtr n)
472 {
473         xmlValidCtxtPtr vcp;

475         if (n->doc == NULL)
476                 uu_die(gettext("Could not validate element\n"));

478         if (n->doc->extSubset == NULL) {
479                 xmlDtdPtr dtd;
480                 dtd = xmlParseDTD(NULL, n->doc->intSubset->SystemID);

482                 if (dtd == NULL) {
483                         uu_die(gettext("Could not parse DTD \"%s\".\n"),
484                             n->doc->intSubset->SystemID);
485                 }

487                 n->doc->extSubset = dtd;
488         }

490         vcp = xmlNewValidCtxt();
491         if (vcp == NULL)
492                 uu_die(gettext("could not allocate memory"));

494         vcp->warning = xmlParserValidityWarning;
495         vcp->error = xmlParserValidityError;

497         if (xmlValidateElement(vcp, n->doc, n) == 0)
498                 uu_die(gettext("Document is not valid.\n"));

500         xmlFreeValidCtxt(vcp);
501 }

503 static int
504 lxml_validate_string_value(scf_type_t type, const char *v)
505 {
506         static scf_value_t *scf_value = NULL;
507         static scf_handle_t *scf_hndl = NULL;

509         if (scf_hndl == NULL && (scf_hndl = scf_handle_create(SCF_VERSION)) ==
510             NULL)
511                 return (-1);

513         if (scf_value == NULL && (scf_value = scf_value_create(scf_hndl)) ==
514             NULL)
515                 return (-1);

517         return (scf_value_set_from_string(scf_value, type, v));
518 }

520 static void
521 lxml_free_str(value_t *val)
522 {
523         free(val->sc_u.sc_string);
```

```
524 }
526 /*
527  * Take a value_t structure and a type and value.  Based on the type
528  * ensure that the value is of that type.  If so store the value in
529  * the correct location of the value_t structure.
530  *
531  * If the value is NULL, the value_t structure will have been created
532  * and the value would have ultimately been stored as a string value
533  * but at the time the type was unknown.  Now the type should be known
534  * so take the type and value from value_t and validate and store
535  * the value correctly if the value is of the stated type.
536  */
537 void
538 lxml_store_value(value_t *v, element_t type, const xmlChar *value)
539 {
540         char *endptr;
541         int fov = 0;
542         scf_type_t scf_type = SCF_TYPE_INVALID;
544         if (value == NULL) {
545                 type = lxml_type_to_element(v->sc_type);
546                 value = (const xmlChar *)v->sc_u.sc_string;
547                 fov = 1;
548         }
550         switch (type) {
551         case SC_COUNT:
552                 /*
553                  * Although an SC_COUNT represents a uint64_t the use
554                  * of a negative value is acceptable due to the usage
555                  * established by inetd(1M).
556                  */
557                 errno = 0;
558                 v->sc_u.sc_count = strtoull((char *)value, &endptr, 10);
559                 if (errno != 0 || endptr == (char *)value || *endptr)
560                         uu_die(gettext("illegal value \"%s\" for "
561                             "%s (%s)\n"), (char *)value,
562                             lxml_prop_types[type],
563                             (errno) ? strerror(errno) :
564                             gettext("Illegal character"));
565                 break;
566         case SC_INTEGER:
567                 errno = 0;
568                 v->sc_u.sc_integer = strtoll((char *)value, &endptr, 10);
569                 if (errno != 0 || *endptr)
570                         uu_die(gettext("illegal value \"%s\" for "
571                             "%s (%s)\n"), (char *)value,
572                             lxml_prop_types[type],
573                             (errno) ? strerror(errno) : "Illegal character");
574                 break;
575         case SC_OPAQUE:
576         case SC_HOST:
577         case SC_HOSTNAME:
578         case SC_NET_ADDR:
579         case SC_NET_ADDR_V4:
580         case SC_NET_ADDR_V6:
581         case SC_FMRI:
582         case SC_URI:
583         case SC_TIME:
584         case SC_ASTRING:
585         case SC_USTRING:
586                 scf_type = lxml_element_to_type(type);
588                 v->sc_u.sc_string = safe_strdup((const char *)value);
 24                 if ((v->sc_u.sc_string = strdup((char *)value)) == NULL)
```



```
 25                         uu_die(gettext("string duplication failed (%s)\n"),
 26                             strerror(errno));
589                 if (lxml_validate_string_value(scf_type,
590                     v->sc_u.sc_string) != 0)
591                         uu_die(gettext("illegal value \"%s\" for "
592                             "%s (%s)\n"), (char *)value,
593                             lxml_prop_types[type],
594                             (scf_error()) ? scf_strerror(scf_error()) :
595                             gettext("Illegal format"));
596                 v->sc_free = lxml_free_str;
597                 break;
598         case SC_BOOLEAN:
599                 v->sc_u.sc_count = lxml_xlate_boolean(value);
600                 break;
601         default:
602                 uu_die(gettext("unknown value type (%d)\n"), type);
603                 break;
604         }
606         /* Free the old value */
607         if (fov && v->sc_free != NULL)
608                 free((char *)value);
609 }
______unchanged_portion_omitted_
1366 static int
1367 lxml_get_entity_stability(entity_t *entity, xmlNodePtr rstr)
1368 {
1369         pgroup_t *pg;
1370         property_t *p;
1371         xmlChar *stabval;
1373         if (((stabval = xmlGetProp(rstr, (xmlChar *)value_attr)) == NULL) ||
1374             (*stabval == 0)) {
1375                 uu_warn(gettext("no stability value found\n"));
1376                 stabval = (xmlChar *)safe_strdup("External");
 814                 stabval = (xmlChar *)strdup("External");
1377         }
1379         pg = internal_pgroup_find_or_create(entity, (char *)scf_pg_general,
1380             (char *)scf_group_framework);
1382         p = internal_property_create(SCF_PROPERTY_ENTITY_STABILITY,
1383             SCF_TYPE_ASTRING, 1, stabval);
1385         return (internal_attach_property(pg, p));
1386 }
______unchanged_portion_omitted_
1437 static void
1438 lxml_get_paramval(pgroup_t *pgrp, const char *propname, xmlNodePtr pval)
1439 {
1440         property_t *p;
1441         char *value;
1442         char *prop;
1444         prop = safe_strdup(propname);
 882         if ((prop = strdup(propname)) == NULL)
 883                 uu_die(gettext("Out of memory.\n"));
1446         value = (char *)xmlGetProp(pval, (xmlChar *)value_attr);
1447         if (value == NULL || *value == '\0')
1448                 uu_die(gettext("property value missing for property '%s/%s'\n"),
1449                     pgrp->sc_pgroup_name, propname);
1450         p = internal_property_create(prop, SCF_TYPE_ASTRING, 1, value);
```

```
1452         (void) internal_attach_property(pgrp, p);
1453 }

1455 static void
1456 lxml_get_parameter(pgroup_t *pgrp, const char *propname, xmlNodePtr param)
1457 {
1458         property_t *p = internal_property_new();

1460         p->sc_property_name = safe_strdup(propname);
 899         if ((p->sc_property_name = strdup(propname)) == NULL)
 900                 uu_die(gettext("Out of memory.\n"));
1461         p->sc_value_type = SCF_TYPE_ASTRING;

1463         (void) lxml_get_value(p, SC_ASTRING, param);

1465         (void) internal_attach_property(pgrp, p);
1466 }
______unchanged_portion_omitted_

1652 /*
1653  * Add a property containing the localized text from the manifest.  The
1654  * property is added to the property group at pg.  The name of the created
1655  * property is based on the format at pn_format.  This is an snprintf(3C)
1656  * format containing a single %s conversion specification.  At conversion
1657  * time, the %s is replaced by the locale designation.
1658  *
1659  * source is the source element and it is only used for error messages.
1660  */
1661 static int
1662 lxml_get_loctext(entity_t *service, pgroup_t *pg, xmlNodePtr loctext,
1663     const char *pn_format, const char *source)
1664 {
1665         int extra;
1666         xmlNodePtr cursor;
1667         xmlChar *val;
1668         char *stripped, *cp;
1669         property_t *p;
1670         char *prop_name;
1671         int r;

1673         if (((val = xmlGetProp(loctext, (xmlChar *)xml_lang_attr)) == NULL) ||
1674             (*val == 0)) {
1675                 if (((val = xmlGetProp(loctext,
1676                     (xmlChar *)lang_attr)) == NULL) || (*val == 0)) {
1677                         val = (xmlChar *)"unknown";
1678                 }
1679         }

1681         _scf_sanitize_locale((char *)val);
1682         prop_name = safe_malloc(max_scf_name_len + 1);
1683         if ((extra = snprintf(prop_name, max_scf_name_len + 1, pn_format,
1684             val)) >= max_scf_name_len + 1) {
1685                 extra -= max_scf_name_len;
1686                 uu_die(gettext("%s attribute is %d characters too long for "
1687                     "%s in %s\n"),
1688                     xml_lang_attr, extra, source, service->sc_name);
1689         }
1690         xmlFree(val);

1692         for (cursor = loctext->xmlChildrenNode; cursor != NULL;
1693             cursor = cursor->next) {
1694                 if (strcmp("text", (const char *)cursor->name) == 0) {
1695                         break;
1696                 } else if (strcmp("comment", (const char *)cursor->name) != 0) {
1697                         uu_die(gettext("illegal element \"%s\" on loctext "
1698                             "element for \"%s\"\n"), cursor->name,
```



```
1699                             service->sc_name);
1700                 }
1701         }

1703         if (cursor == NULL) {
1704                 uu_die(gettext("loctext element has no content for \"%s\"\n"),
1705                     service->sc_name);
1706         }

1708         /*
1709          * Remove leading and trailing whitespace.
1710          */
1711         stripped = safe_strdup((const char *)cursor->content);
1151         if ((stripped = strdup((const char *)cursor->content)) == NULL)
1152                 uu_die(gettext("Out of memory\n"));

1713         for (; isspace(*stripped); stripped++)
1714                 ;
1715         for (cp = stripped + strlen(stripped) - 1; isspace(*cp); cp--)
1716                 ;
1717         *(cp + 1) = '\0';

1719         p = internal_property_create(prop_name, SCF_TYPE_USTRING, 1,
1720             stripped);

1722         r = internal_attach_property(pg, p);
1723         if (r != 0) {
1724                 internal_property_free(p);
1725                 free(prop_name);
1726         }

1728         return (r);
1729 }
______unchanged_portion_omitted_

1941 static char *
1942 lxml_label_to_groupname(const char *prefix, const char *in)
1943 {
1944         char *out, *cp;
1945         size_t len, piece_len;

1947         out = uu_zalloc(2 * max_scf_name_len + 1);
1388         out = uu_zalloc(2 * scf_limit(SCF_LIMIT_MAX_NAME_LENGTH) + 1);
1948         if (out == NULL)
1949                 return (NULL);

1951         (void) strlcpy(out, prefix, 2 * max_scf_name_len + 1);
1392         (void) strcpy(out, prefix);
1393         (void) strcat(out, in);

1953         len = strlcat(out, in, 2 * max_scf_name_len + 1);
1395         len = strlen(out);
1954         if (len > max_scf_name_len) {
1955                 /* Use the first half and the second half. */
1956                 piece_len = (max_scf_name_len - 2) / 2;

1958                 (void) strncpy(out + piece_len, "..", 2);

1960                 (void) strcpy(out + piece_len + 2, out + (len - piece_len));

1404                 len = strlen(out);
1961         }

1963         /*
1964          * Translate non-property characters to '_'.
1965          */
```

```
1966         for (cp = out; *cp != '\0'; ++cp) {
1967                 if (!(isalnum(*cp) || *cp == '_' || *cp == '-'))
1968                         *cp = '_';
1969         }

1415         *cp = '\0';

1971         return (out);
1972 }
```

*unchanged_portion_omitted_*

```
2009 /*
2010  * If p points to a null pointer, create an internal_separators property
2011  * saving the address at p.  For each character at seps create a property
2012  * value and attach it to the property at p.
2013  */
2014 static void
2015 seps_to_prop_values(property_t **p, xmlChar *seps)
2016 {
2017         value_t *v;
2018         char val_str[2];

2020         if (*p == NULL) {
2021                 *p = internal_property_new();
2022                 (*p)->sc_property_name =
2023                     (char *)SCF_PROPERTY_INTERNAL_SEPARATORS;
2024                 (*p)->sc_value_type = SCF_TYPE_ASTRING;
2025         }

2027         /* Add the values to the property's list. */
2028         val_str[1] = 0;         /* Terminate the string. */
2029         for (; *seps != 0; seps++) {
2030                 v = internal_value_new();
2031                 v->sc_type = (*p)->sc_value_type;
2032                 v->sc_free = lxml_free_str;
2033                 val_str[0] = *seps;
2034                 v->sc_u.sc_string = safe_strdup(val_str);
1480                 v->sc_u.sc_string = strdup(val_str);
1481                 if (v->sc_u.sc_string == NULL)
1482                         uu_die(gettext("Out of memory\n"));
2035                 internal_attach_value(*p, v);
2036         }
2037 }
```

*unchanged_portion_omitted_*

```
2074 static int
2075 lxml_get_tm_manpage(entity_t *service, xmlNodePtr manpage)
2076 {
2077         pgroup_t *pg;
2078         char *pgname;
2079         char *name;
2080 #endif /* ! codereview */
2081         xmlChar *title;
2082         xmlChar *section;
2083 #endif /* ! codereview */

2085         /*
2086          * Fetch title and section attributes, convert to something sanitized,
2087          * and create property group.
1527          * Fetch title attribute, convert to something sanitized, and create
1528          * property group.
2088          */
2089         title = xmlGetProp(manpage, (xmlChar *)title_attr);
2090         if (title == NULL)
2091                 return (-1);
2092         section = xmlGetProp(manpage, (xmlChar *)section_attr);
```



```
2093         if (section == NULL) {
1531         pgname = (char *)lxml_label_to_groupname(SCF_PG_TM_MAN_PREFIX,
1532             (const char *)title);
2094                 xmlFree(title);
2095                 return (-1);
2096         }
2097 #endif /* ! codereview */

2099         name = safe_malloc(max_scf_name_len + 1);

2101         /* Find existing property group with underscore separators */
2102         (void) snprintf(name, max_scf_name_len + 1, "%s_%s", title, section);
2103         pgname = lxml_label_to_groupname(SCF_PG_TM_MAN_PREFIX, name);
2104         pg = internal_pgroup_find(service, pgname, SCF_GROUP_TEMPLATE);

2106         uu_free(pgname);
2107         (void) snprintf(name, max_scf_name_len + 1, "%s%s", title, section);
2108         pgname = lxml_label_to_groupname(SCF_PG_TM_MAN_PREFIX, name);

2110         if (pg == NULL) {
2111 #endif /* ! codereview */
2112                 pg = internal_pgroup_find_or_create(service, pgname,
2113                     SCF_GROUP_TEMPLATE);
2114         } else {
2115                 /* Rename property group */
2116                 free((char *)pg->sc_pgroup_name);
2117                 pg->sc_pgroup_name = safe_strdup(pgname);
2118         }

2120         uu_free(pgname);
2121         free(name);
2122         xmlFree(section);
2123         xmlFree(title);

1534             (char *)SCF_GROUP_TEMPLATE);

2126         /*
2127          * Each attribute is an astring property within the group.
2128          */
2129         if (new_str_prop_from_attr(pg, SCF_PROPERTY_TM_TITLE,
2130             SCF_TYPE_ASTRING, manpage, title_attr) != 0 ||
2131             new_str_prop_from_attr(pg, SCF_PROPERTY_TM_SECTION,
2132             SCF_TYPE_ASTRING, manpage, section_attr) != 0 ||
2133             new_str_prop_from_attr(pg, SCF_PROPERTY_TM_MANPATH,
2134             SCF_TYPE_ASTRING, manpage, manpath_attr) != 0)
2135                 return (-1);

2137         return (0);
2138 }

2140 static int
2141 lxml_get_tm_doclink(entity_t *service, xmlNodePtr doc_link)
2142 {
2143         pgroup_t *pg;
2144         char *pgname;
2145         xmlChar *name;

2147         /*
2148          * Fetch name attribute, convert name to something sanitized, and create
2149          * property group.
2150          */
2151         name = xmlGetProp(doc_link, (xmlChar *)name_attr);
2152         if (name == NULL)
2153                 return (-1);
2154 #endif /* ! codereview */
```

```
2156         pgname = (char *)lxml_label_to_groupname(SCF_PG_TM_DOC_PREFIX,
2157             (const char *)name);

2159         pg = internal_pgroup_find_or_create(service, pgname,
2160             (char *)SCF_GROUP_TEMPLATE);

2162         uu_free(pgname);
2163 #endif /* ! codereview */
2164         xmlFree(name);

2166         /*
2167          * Each attribute is an astring property within the group.
2168          */
2169         if (new_str_prop_from_attr(pg, SCF_PROPERTY_TM_NAME, SCF_TYPE_ASTRING,
2170             doc_link, name_attr) != 0 ||
2171             new_str_prop_from_attr(pg, SCF_PROPERTY_TM_URI, SCF_TYPE_ASTRING,
2172             doc_link, uri_attr) != 0)
2173                 return (-1);

2175         return (0);
2176 }

2178 static int
2179 lxml_get_tm_documentation(entity_t *service, xmlNodePtr documentation)
2180 {
2181         xmlNodePtr cursor;

2183         for (cursor = documentation->xmlChildrenNode; cursor != NULL;
2184             cursor = cursor->next) {
2185                 if (lxml_ignorable_block(cursor))
2186                         continue;

2188                 switch (lxml_xlate_element(cursor->name)) {
2189                 case SC_MANPAGE:
2190                         (void) lxml_get_tm_manpage(service, cursor);
2191                         break;
2192                 case SC_DOC_LINK:
2193                         (void) lxml_get_tm_doclink(service, cursor);
2194                         break;
2195                 default:
2196                         uu_die(gettext("illegal element \"%s\" on template "
2197                             "for service \"%s\"\n"),
2198                             cursor->name, service->sc_name);
2199                 }
2200         }

2202         return (0);
2203 }

2205 static int
2206 lxml_get_prop_pattern_attributes(pgroup_t *pg, xmlNodePtr cursor)
2207 {
2208         if (new_opt_str_prop_from_attr(pg, SCF_PROPERTY_TM_NAME,
2209             SCF_TYPE_ASTRING, cursor, name_attr, NULL) != 0) {
2210                 return (-1);
2211         }
2212         if (new_opt_str_prop_from_attr(pg, SCF_PROPERTY_TM_TYPE,
2213             SCF_TYPE_ASTRING, cursor, type_attr, "") != 0) {
2214                 return (-1);
2215         }
2216         if (new_bool_prop_from_attr(pg, SCF_PROPERTY_TM_REQUIRED, cursor,
2217             required_attr) != 0)
2218                 return (-1);
2219         return (0);
2220 }
```



```
2222 static int
2223 lxml_get_tm_include_values(entity_t *service, pgroup_t *pg,
2224     xmlNodePtr include_values, const char *prop_name)
2225 {
2226         boolean_t attach_to_pg = B_FALSE;
2227         property_t *p;
2228         int r = 0;
2229         char *type;

2231         /* Get the type attribute of the include_values element. */
2232         type = (char *)xmlGetProp(include_values, (const xmlChar *)type_attr);
2233         if ((type == NULL) || (*type == 0)) {
2234                 uu_die(gettext("%s element requires a %s attribute in the %s "
2235                     "service.\n"), include_values->name, type_attr,
2236                     service->sc_name);
2237         }

2239         /* Add the type to the values of the prop_name property. */
2240         p = internal_property_find(pg, prop_name);
2241         if (p == NULL)
2242                 attach_to_pg = B_TRUE;
2243         astring_prop_value(&p, prop_name, type, B_FALSE);
2244         if (attach_to_pg == B_TRUE) {
2245                 r = internal_attach_property(pg, p);
2246                 if (r != 0)
2247                         internal_property_free(p);
2248         }
2249         return (r);
2250 }

2252 #define RC_MIN          0
2253 #define RC_MAX          1
2254 #define RC_COUNT        2

2256 /*
2257  * Verify that the strings at min and max are valid numeric strings.  Also
2258  * verify that max is numerically >= min.
2259  *
2260  * 0 is returned if the range is valid, and -1 is returned if it is not.
2261  */
2262 static int
2263 verify_range(entity_t *service, xmlNodePtr range, char *min, char *max)
2264 {
2265         char *c;
2266         int i;
2267         int is_signed = 0;
2268         int inverted = 0;
2269         const char *limit[RC_COUNT];
2270         char *strings[RC_COUNT];
2271         uint64_t urange[RC_COUNT];      /* unsigned range. */
2272         int64_t srange[RC_COUNT];       /* signed range. */

2274         strings[RC_MIN] = min;
2275         strings[RC_MAX] = max;
2276         limit[RC_MIN] = min_attr;
2277         limit[RC_MAX] = max_attr;

2279         /* See if the range is signed. */
2280         for (i = 0; (i < RC_COUNT) && (is_signed == 0); i++) {
2281                 c = strings[i];
2282                 while (isspace(*c)) {
2283                         c++;
2284                 }
2285                 if (*c == '-')
2286                         is_signed = 1;
2287         }
```

```
2289         /* Attempt to convert the strings. */
2290         for (i = 0; i < RC_COUNT; i++) {
2291                 errno = 0;
2292                 if (is_signed) {
2293                         srange[i] = strtoll(strings[i], &c, 0);
2294                 } else {
2295                         urange[i] = strtoull(strings[i], &c, 0);
2296                 }
2297                 if ((errno != 0) || (c == strings[i]) || (*c != 0)) {
2298                         /* Conversion failed. */
2299                         uu_die(gettext("Unable to convert %s for the %s "
2300                             "element in service %s.\n"), limit[i],
2301                             (char *)range->name, service->sc_name);
2302                 }
2303         }

2305         /* Make sure that min is <= max */
2306         if (is_signed) {
2307                 if (srange[RC_MAX] < srange[RC_MIN])
2308                         inverted = 1;
2309         } else {
2310                 if (urange[RC_MAX] < urange[RC_MIN])
2311                         inverted = 1;
2312         }
2313         if (inverted != 0) {
2314                 semerr(gettext("Maximum less than minimum for the %s element "
2315                     "in service %s.\n"), (char *)range->name,
2316                     service->sc_name);
2317                 return (-1);
2318         }

2320         return (0);
2321 }

2323 /*
2324  * This, function creates a property named prop_name.  The range element
2325  * should have two attributes -- min and max.  The property value then
2326  * becomes the concatenation of their value separated by a comma.  The
2327  * property is then attached to the property group at pg.
2328  *
2329  * If pg already contains a property with a name of prop_name, it is only
2330  * necessary to create a new value and attach it to the existing property.
2331  */
2332 static int
2333 lxml_get_tm_range(entity_t *service, pgroup_t *pg, xmlNodePtr range,
2334     const char *prop_name)
2335 {
2336         boolean_t attach_to_pg = B_FALSE;
2337         char *max;
2338         char *min;
2339         property_t *p;
2340         char *prop_value;
2341         int r = 0;

2343         /* Get max and min from the XML description. */
2344         max = (char *)xmlGetProp(range, (xmlChar *)max_attr);
2345         if ((max == NULL) || (*max == 0)) {
2346                 uu_die(gettext("%s element is missing the %s attribute in "
2347                     "service %s.\n"), (char *)range->name, max_attr,
2348                     service->sc_name);
2349         }
2350         min = (char *)xmlGetProp(range, (xmlChar *)min_attr);
2351         if ((min == NULL) || (*min == 0)) {
2352                 uu_die(gettext("%s element is missing the %s attribute in "
2353                     "service %s.\n"), (char *)range->name, min_attr,
```



```
2354                     service->sc_name);
2355         }
2356         if (verify_range(service, range, min, max) != 0) {
2357                 xmlFree(min);
2358                 xmlFree(max);
2359                 return (-1);
2360         }

2362         /* Property value is concatenation of min and max. */
2363         prop_value = safe_malloc(max_scf_value_len + 1);
2364         if (snprintf(prop_value, max_scf_value_len + 1, "%s,%s", min, max) >=
2365             max_scf_value_len + 1) {
2366                 uu_die(gettext("min and max are too long for the %s element "
2367                     "of %s.\n"), (char *)range->name, service->sc_name);
2368         }
2369         xmlFree(min);
2370         xmlFree(max);

2372         /*
2373          * If necessary create the property and attach it to the property
2374          * group.
2375          */
2376         p = internal_property_find(pg, prop_name);
2377         if (p == NULL)
2378                 attach_to_pg = B_TRUE;
2379         astring_prop_value(&p, prop_name, prop_value, B_TRUE);
2380         if (attach_to_pg == B_TRUE) {
2381                 r = internal_attach_property(pg, p);
2382                 if (r != 0) {
2383                         internal_property_free(p);
2384                 }
2385         }
2386         return (r);
2387 }

2389 /*
2390  * Determine how many plain characters are represented by count Base32
2391  * encoded characters.  5 plain text characters are converted to 8 Base32
2392  * characters.
2393  */
2394 static size_t
2395 encoded_count_to_plain(size_t count)
2396 {
2397         return (5 * ((count + 7) / 8));
2398 }

2400 /*
2401  * The value element contains 0 or 1 common_name element followed by 0 or 1
2402  * description element.  It also has a required attribute called "name".
2403  * The common_name and description are stored as property values in pg.
2404  * The property names are:
2405  *      value_<name>_common_name_<lang>
2406  *      value_<name>_description_<lang>
2407  *
2408  * The <name> portion of the preceeding proper names requires more
2409  * explanation.  Ideally it would just the name attribute of this value
2410  * element.  Unfortunately, the name attribute can contain characters that
2411  * are not legal in a property name.  Thus, we base 32 encode the name
2412  * attribute and use that for <name>.
2413  *
2414  * There are cases where the caller needs to know the name, so it is
2415  * returned through the name_value pointer if it is not NULL.
2416  *
2417  * Parameters:
2418  *      service -       Information about the service that is being
2419  *                      processed.  This function only uses this parameter
```

```
 *                      for producing error messages.
 *
 *      pg -            The property group to receive the newly created
 *                      properties.
 *
 *      value -         Pointer to the value element in the XML tree.
 *
 *      name_value -    Address to receive the value of the name attribute.
 *                      The caller must free the memory.
 */
static int
lxml_get_tm_value_element(entity_t *service, pgroup_t *pg, xmlNodePtr value,
    char **name_value)
{
        char *common_name_fmt;
        xmlNodePtr cursor;
        char *description_fmt;
        char *encoded_value = NULL;
        size_t extra;
        char *value_name;
        int r = 0;

        common_name_fmt = safe_malloc(max_scf_name_len + 1);
        description_fmt = safe_malloc(max_scf_name_len + 1);

        /*
         * Get the value of our name attribute, so that we can use it to
         * construct property names.
         */
        value_name = (char *)xmlGetProp(value, (xmlChar *)name_attr);
        /* The value name must be present, but it can be empty. */
        if (value_name == NULL) {
                uu_die(gettext("%s element requires a %s attribute in the %s "
                    "service.\n"), (char *)value->name, name_attr,
                    service->sc_name);
        }

        /*
         * The value_name may contain characters that are not valid in in a
         * property name.  So we will encode value_name and then use the
         * encoded value in the property name.
         */
        encoded_value = safe_malloc(max_scf_name_len + 1);
        if (scf_encode32(value_name, strlen(value_name), encoded_value,
            max_scf_name_len + 1, &extra, SCF_ENCODE32_PAD) != 0) {
                extra = encoded_count_to_plain(extra - max_scf_name_len);
                uu_die(gettext("Constructed property name is %u characters "
                    "too long for value \"%s\" in the %s service.\n"),
                    extra, value_name, service->sc_name);
        }
        if ((extra = snprintf(common_name_fmt, max_scf_name_len + 1,
            VALUE_COMMON_NAME_FMT, SCF_PROPERTY_TM_VALUE_PREFIX,
            encoded_value)) >= max_scf_name_len + 1) {
                extra = encoded_count_to_plain(extra - max_scf_name_len);
                uu_die(gettext("Name attribute is "
                    "%u characters too long for %s in service %s\n"),
                    extra, (char *)value->name, service->sc_name);
        }
        if ((extra = snprintf(description_fmt, max_scf_name_len + 1,
            VALUE_DESCRIPTION_FMT, SCF_PROPERTY_TM_VALUE_PREFIX,
            encoded_value)) >= max_scf_name_len + 1) {
                extra = encoded_count_to_plain(extra - max_scf_name_len);
                uu_die(gettext("Name attribute is "
                    "%u characters too long for %s in service %s\n"),
                    extra, (char *)value->name, service->sc_name);
        }
```



```
        for (cursor = value->xmlChildrenNode;
            cursor != NULL;
            cursor = cursor->next) {
                if (lxml_ignorable_block(cursor))
                        continue;
                switch (lxml_xlate_element(cursor->name)) {
                case SC_COMMON_NAME:
                        r = lxml_get_all_loctext(service, pg, cursor,
                            common_name_fmt, (const char *)cursor->name);
                        break;
                case SC_DESCRIPTION:
                        r = lxml_get_all_loctext(service, pg, cursor,
                            description_fmt, (const char *)cursor->name);
                        break;
                default:
                        uu_die(gettext("\"%s\" is an illegal element in %s "
                            "of service %s\n"), (char *)cursor->name,
                            (char *)value->name, service->sc_name);
                }
                if (r != 0)
                        break;
        }

        free(description_fmt);
        free(common_name_fmt);
        if (r == 0) {
                *name_value = safe_strdup(value_name);
        }
        xmlFree(value_name);
        free(encoded_value);
        return (r);
}

static int
lxml_get_tm_choices(entity_t *service, pgroup_t *pg, xmlNodePtr choices)
{
        xmlNodePtr cursor;
        char *name_value;
        property_t *name_prop = NULL;
        int r = 0;

        for (cursor = choices->xmlChildrenNode;
            (cursor != NULL) && (r == 0);
            cursor = cursor->next) {
                if (lxml_ignorable_block(cursor))
                        continue;
                switch (lxml_xlate_element(cursor->name)) {
                case SC_INCLUDE_VALUES:
                        (void) lxml_get_tm_include_values(service, pg, cursor,
                            SCF_PROPERTY_TM_CHOICES_INCLUDE_VALUES);
                        break;
                case SC_RANGE:
                        r = lxml_get_tm_range(service, pg, cursor,
                            SCF_PROPERTY_TM_CHOICES_RANGE);
                        if (r != 0)
                                goto out;
                        break;
                case SC_VALUE:
                        r = lxml_get_tm_value_element(service, pg, cursor,
                            &name_value);
                        if (r == 0) {
                                /*
                                 * There is no need to free the memory
                                 * associated with name_value, because the
                                 * property value will end up pointing to
```

```
2552                                * the memory.
2553                                */
2554                               astring_prop_value(&name_prop,
2555                                   SCF_PROPERTY_TM_CHOICES_NAME, name_value,
2556                                   B_TRUE);
2557                       } else {
2558                               goto out;
2559                       }
2560                       break;
2561               default:
2562                       uu_die(gettext("%s is an invalid element of "
2563                           "choices for service %s.\n"),  cursor->name,
2564                           service->sc_name);
2565               }
2566       }

2568 out:
2569       /* Attach the name property if we created one. */
2570       if ((r == 0) && (name_prop != NULL)) {
2571               r = internal_attach_property(pg, name_prop);
2572       }
2573       if ((r != 0) && (name_prop != NULL)) {
2574               internal_property_free(name_prop);
2575       }

2577       return (r);
2578 }

2580 static int
2581 lxml_get_tm_constraints(entity_t *service, pgroup_t *pg, xmlNodePtr constraints)
2582 {
2583       xmlNodePtr cursor;
2584       char *name_value;
2585       property_t *name_prop = NULL;
2586       int r = 0;

2588       for (cursor = constraints->xmlChildrenNode;
2589           (cursor != NULL) && (r == 0);
2590           cursor = cursor->next) {
2591               if (lxml_ignorable_block(cursor))
2592                       continue;
2593               switch (lxml_xlate_element(cursor->name)) {
2594               case SC_RANGE:
2595                       r = lxml_get_tm_range(service, pg, cursor,
2596                           SCF_PROPERTY_TM_CONSTRAINT_RANGE);
2597                       if (r != 0)
2598                               goto out;
2599                       break;
2600               case SC_VALUE:
2601                       r = lxml_get_tm_value_element(service, pg, cursor,
2602                           &name_value);
2603                       if (r == 0) {
2604                               /*
2605                                * There is no need to free the memory
2606                                * associated with name_value, because the
2607                                * property value will end up pointing to
2608                                * the memory.
2609                                */
2610                               astring_prop_value(&name_prop,
2611                                   SCF_PROPERTY_TM_CONSTRAINT_NAME, name_value,
2612                                   B_TRUE);
2613                       } else {
2614                               goto out;
2615                       }
2616                       break;
2617               default:
```



```
2618                       uu_die(gettext("%s is an invalid element of "
2619                           "constraints for service %s.\n"),  cursor->name,
2620                           service->sc_name);
2621               }
2622       }

2624 out:
2625       /* Attach the name property if we created one. */
2626       if ((r == 0) && (name_prop != NULL)) {
2627               r = internal_attach_property(pg, name_prop);
2628       }
2629       if ((r != 0) && (name_prop != NULL)) {
2630               internal_property_free(name_prop);
2631       }

2633       return (r);
2634 }

2636 /*
2637  * The values element contains one or more value elements.
2638  */
2639 static int
2640 lxml_get_tm_values(entity_t *service, pgroup_t *pg, xmlNodePtr values)
2641 {
2642       xmlNodePtr cursor;
2643       char *name_value;
2644       property_t *name_prop = NULL;
2645       int r = 0;

2647       for (cursor = values->xmlChildrenNode;
2648           (cursor != NULL) && (r == 0);
2649           cursor = cursor->next) {
2650               if (lxml_ignorable_block(cursor))
2651                       continue;
2652               if (lxml_xlate_element(cursor->name) != SC_VALUE) {
2653                       uu_die(gettext("\"%s\" is an illegal element in the "
2654                           "%s element of %s\n"), (char *)cursor->name,
2655                           (char *)values->name, service->sc_name);
2656               }
2657               r = lxml_get_tm_value_element(service, pg, cursor, &name_value);
2658               if (r == 0) {
2659                       /*
2660                        * There is no need to free the memory
2661                        * associated with name_value, because the
2662                        * property value will end up pointing to
2663                        * the memory.
2664                        */
2665                       astring_prop_value(&name_prop,
2666                           SCF_PROPERTY_TM_VALUES_NAME, name_value,
2667                           B_TRUE);
2668               }
2669       }

2671       /* Attach the name property if we created one. */
2672       if ((r == 0) && (name_prop != NULL)) {
2673               r = internal_attach_property(pg, name_prop);
2674       }
2675       if ((r != 0) && (name_prop != NULL)) {
2676               internal_property_free(name_prop);
2677       }

2679       return (r);
2680 }

2682 /*
2683  * This function processes a prop_pattern element within a pg_pattern XML
```

```
 * element.  First it creates a property group to hold the prop_pattern
 * information.  The name of this property group is the concatenation of:
 *      - SCF_PG_TM_PROP_PATTERN_PREFIX
 *      - The unique part of the property group name of the enclosing
 *        pg_pattern.  The property group name of the enclosing pg_pattern
 *        is passed to us in pgpat_name.  The unique part, is the part
 *        following SCF_PG_TM_PG_PATTERN_PREFIX.
 *      - The name of this prop_pattern element.
 *
 * After creating the property group, the prop_pattern attributes are saved
 * as properties in the PG.  Finally, the prop_pattern elements are
 * processed and added to the PG.
 */
static int
lxml_get_tm_prop_pattern(entity_t *service, xmlNodePtr prop_pattern,
    const char *pgpat_name)
{
        xmlNodePtr cursor;
        int extra;
        pgroup_t *pg;
        property_t *p;
        char *pg_name;
        size_t prefix_len;
        xmlChar *prop_pattern_name;
        int r;
        const char *unique;
        value_t *v;

        /* Find the unique part of the pg_pattern property group name. */
        prefix_len = strlen(SCF_PG_TM_PG_PAT_BASE);
        assert(strncmp(pgpat_name, SCF_PG_TM_PG_PAT_BASE, prefix_len) == 0);
        unique = pgpat_name + prefix_len;

        /*
         * We need to get the value of the name attribute first.  The
         * prop_pattern name as well as the name of the enclosing
         * pg_pattern both constitute part of the name of the property
         * group that we will create.
         */
        prop_pattern_name = xmlGetProp(prop_pattern, (xmlChar *)name_attr);
        if ((prop_pattern_name == NULL) || (*prop_pattern_name == 0)) {
                semerr(gettext("prop_pattern name is missing for %s\n"),
                    service->sc_name);
                return (-1);
        }
        if (uu_check_name((const char *)prop_pattern_name,
            UU_NAME_DOMAIN) != 0) {
                semerr(gettext("prop_pattern name, \"%s\", for %s is not "
                    "valid.\n"), prop_pattern_name, service->sc_name);
                xmlFree(prop_pattern_name);
                return (-1);
        }
        pg_name = safe_malloc(max_scf_name_len + 1);
        if ((extra = snprintf(pg_name, max_scf_name_len + 1, "%s%s_%s",
            SCF_PG_TM_PROP_PATTERN_PREFIX, unique,
            (char *)prop_pattern_name)) >= max_scf_name_len + 1) {
                uu_die(gettext("prop_pattern name, \"%s\", for %s is %d "
                    "characters too long\n"), (char *)prop_pattern_name,
                    service->sc_name, extra - max_scf_name_len);
        }

        /*
         * Create the property group, the property referencing the pg_pattern
         * name, and add the prop_pattern attributes to the property group.
         */
        pg = internal_pgroup_create_strict(service, pg_name,
```



```
            SCF_GROUP_TEMPLATE_PROP_PATTERN);
        if (pg == NULL) {
                uu_die(gettext("Property group for prop_pattern, \"%s\", "
                    "already exists in %s\n"), prop_pattern_name,
                    service->sc_name);
        }

        p = internal_property_create(SCF_PROPERTY_TM_PG_PATTERN,
            SCF_TYPE_ASTRING, 1, safe_strdup(pgpat_name));
        /*
         * Unfortunately, internal_property_create() does not set the free
         * function for the value, so we'll set it now.
         */
        v = uu_list_first(p->sc_property_values);
        v->sc_free = lxml_free_str;
        if (internal_attach_property(pg, p) != 0)
                internal_property_free(p);


        r = lxml_get_prop_pattern_attributes(pg, prop_pattern);
        if (r != 0)
                goto out;

        /*
         * Now process the elements of prop_pattern
         */
        for (cursor = prop_pattern->xmlChildrenNode;
            cursor != NULL;
            cursor = cursor->next) {
                if (lxml_ignorable_block(cursor))
                        continue;

                switch (lxml_xlate_element(cursor->name)) {
                case SC_CARDINALITY:
                        r = lxml_get_tm_cardinality(service, pg, cursor);
                        if (r != 0)
                                goto out;
                        break;
                case SC_CHOICES:
                        r = lxml_get_tm_choices(service, pg, cursor);
                        if (r != 0)
                                goto out;
                        break;
                case SC_COMMON_NAME:
                        (void) lxml_get_all_loctext(service, pg, cursor,
                            COMMON_NAME_FMT, (const char *)cursor->name);
                        break;
                case SC_CONSTRAINTS:
                        r = lxml_get_tm_constraints(service, pg, cursor);
                        if (r != 0)
                                goto out;
                        break;
                case SC_DESCRIPTION:
                        (void) lxml_get_all_loctext(service, pg, cursor,
                            DESCRIPTION_FMT, (const char *)cursor->name);
                        break;
                case SC_INTERNAL_SEPARATORS:
                        r = lxml_get_tm_internal_seps(service, pg, cursor);
                        if (r != 0)
                                goto out;
                        break;
                case SC_UNITS:
                        (void) lxml_get_all_loctext(service, pg, cursor,
                            UNITS_FMT, "units");
                        break;
                case SC_VALUES:
```

```
2816                        (void) lxml_get_tm_values(service, pg, cursor);
2817                        break;
2818                case SC_VISIBILITY:
2819                        /*
2820                         * The visibility element is empty, so we only need
2821                         * to proccess the value attribute.
2822                         */
2823                        (void) new_str_prop_from_attr(pg,
2824                            SCF_PROPERTY_TM_VISIBILITY, SCF_TYPE_ASTRING,
2825                            cursor, value_attr);
2826                        break;
2827                default:
2828                        uu_die(gettext("illegal element \"%s\" in prop_pattern "
2829                            "for service \"%s\"\n"), cursor->name,
2830                            service->sc_name);
2831                }
2832        }

2834 out:
2835        xmlFree(prop_pattern_name);
2836        free(pg_name);
2837        return (r);
2838 }

2840 /*
2841  * Get the pg_pattern attributes and save them as properties in the
2842  * property group at pg.  The pg_pattern element accepts four attributes --
2843  * name, type, required and target.
2844  */
2845 static int
2846 lxml_get_pg_pattern_attributes(pgroup_t *pg, xmlNodePtr cursor)
2847 {
2848        if (new_opt_str_prop_from_attr(pg, SCF_PROPERTY_TM_NAME,
2849            SCF_TYPE_ASTRING, cursor, name_attr, NULL) != 0) {
2850                return (-1);
2851        }
2852        if (new_opt_str_prop_from_attr(pg, SCF_PROPERTY_TM_TYPE,
2853            SCF_TYPE_ASTRING, cursor, type_attr, NULL) != 0) {
2854                return (-1);
2855        }
2856        if (new_opt_str_prop_from_attr(pg, SCF_PROPERTY_TM_TARGET,
2857            SCF_TYPE_ASTRING, cursor, target_attr, NULL) != 0) {
2858                return (-1);
2859        }
2860        if (new_bool_prop_from_attr(pg, SCF_PROPERTY_TM_REQUIRED, cursor,
2861            required_attr) != 0)
2862                return (-1);
2863        return (0);
2864 }

2866 /*
2867  * There are several restrictions on the pg_pattern attributes that cannot
2868  * be specifed in the service bundle DTD.  This function verifies that
2869  * those restrictions have been satisfied.  The restrictions are:
2870  *
2871  *      - The target attribute may have a value of "instance" only when the
2872  *        template block is in a service declaration.
2873  *
2874  *      - The target attribute may have a value of "delegate" only when the
2875  *        template block applies to a restarter.
2876  *
2877  *      - The target attribute may have a value of "all" only when the
2878  *        template block applies to the master restarter.
2879  *
2880  * The function returns 0 on success and -1 on failure.
2881  */
```



```
2882 static int
2883 verify_pg_pattern_attributes(entity_t *s, pgroup_t *pg)
2884 {
2885        int is_restarter;
2886        property_t *target;
2887        value_t *v;

2889        /* Find the value of the target property. */
2890        target = internal_property_find(pg, SCF_PROPERTY_TM_TARGET);
2891        if (target == NULL) {
2892                uu_die(gettext("pg_pattern is missing the %s attribute "
2893                    "in %s\n"), target_attr, s->sc_name);
2894                return (-1);
2895        }
2896        v = uu_list_first(target->sc_property_values);
2897        assert(v != NULL);
2898        assert(v->sc_type == SCF_TYPE_ASTRING);

2900        /*
2901         * If target has a value of instance, the template must be in a
2902         * service object.
2903         */
2904        if (strcmp(v->sc_u.sc_string, "instance") == 0) {
2905                if (s->sc_etype != SVCCFG_SERVICE_OBJECT) {
2906                        uu_warn(gettext("pg_pattern %s attribute may only "
2907                            "have a value of \"instance\" when it is in a "
2908                            "service declaration.\n"), target_attr);
2909                        return (-1);
2910                }
2911        }

2913        /*
2914         * If target has a value of "delegate", the template must be in a
2915         * restarter.
2916         */
2917        if (strcmp(v->sc_u.sc_string, "delegate") == 0) {
2918                is_restarter = 0;
2919                if ((s->sc_etype == SVCCFG_SERVICE_OBJECT) &&
2920                    (s->sc_u.sc_service.sc_service_type == SVCCFG_RESTARTER)) {
2921                        is_restarter = 1;
2922                }
2923                if ((s->sc_etype == SVCCFG_INSTANCE_OBJECT) &&
2924                    (s->sc_parent->sc_u.sc_service.sc_service_type ==
2925                    SVCCFG_RESTARTER)) {
2926                        is_restarter = 1;
2927                }
2928                if (is_restarter == 0) {
2929                        uu_warn(gettext("pg_pattern %s attribute has a "
2930                            "value of \"delegate\" but is not in a "
2931                            "restarter service\n"), target_attr);
2932                        return (-1);
2933                }
2934        }

2936        /*
2937         * If target has a value of "all", the template must be in the
2938         * global (SCF_SERVICE_GLOBAL) service.
2939         */
2940        if (strcmp(v->sc_u.sc_string, all_value) == 0) {
2941                if (s->sc_etype != SVCCFG_SERVICE_OBJECT) {
2942                        uu_warn(gettext("pg_pattern %s attribute has a "
2943                            "value of \"%s\" but is not in a "
2944                            "service entity.\n"), target_attr, all_value);
2945                        return (-1);
2946                }
2947                if (strcmp(s->sc_fmri, SCF_SERVICE_GLOBAL) != 0) {
```

```
2948                        uu_warn(gettext("pg_pattern %s attribute has a "
2949                            "value of \"%s\" but is in the \"%s\" service.  "
2950                            "pg_patterns with target \"%s\" are only allowed "
2951                            "in the global service.\n"),
2952                            target_attr, all_value, s->sc_fmri, all_value);
2953                        return (-1);
2954                }
2955        }

2957        return (0);
2958 }

2960 static int
2961 lxml_get_tm_pg_pattern(entity_t *service, xmlNodePtr pg_pattern)
2962 {
2963        xmlNodePtr cursor;
2964        int out_len;
2965        xmlChar *name;
2966        pgroup_t *pg = NULL;
2967        char *pg_name;
2968        int r = -1;
2969        xmlChar *type;

2971        pg_name = safe_malloc(max_scf_name_len + 1);

2973        /*
2974         * Get the name and type attributes.  Their presence or absence
2975         * determines whcih prefix we will use for the property group name.
2976         * There are four cases -- neither attribute is present, both are
2977         * present, only name is present or only type is present.
2978         */
2979        name = xmlGetProp(pg_pattern, (xmlChar *)name_attr);
2980        type = xmlGetProp(pg_pattern, (xmlChar *)type_attr);
2981        if ((name == NULL) || (*name == 0)) {
2982                if ((type == NULL) || (*type == 0)) {
2983                        /* PG name contains only the prefix in this case */
2984                        if (strlcpy(pg_name, SCF_PG_TM_PG_PATTERN_PREFIX,
2985                            max_scf_name_len + 1) >= max_scf_name_len + 1) {
2986                                uu_die(gettext("Unable to create pg_pattern "
2987                                    "property for %s\n"), service->sc_name);
2988                        }
2989                } else {
2990                        /*
2991                         * If we have a type and no name, the type becomes
2992                         * part of the pg_pattern property group name.
2993                         */
2994                        if ((out_len = snprintf(pg_name, max_scf_name_len + 1,
2995                            "%s%s", SCF_PG_TM_PG_PATTERN_T_PREFIX, type)) >=
2996                            max_scf_name_len + 1) {
2997                                uu_die(gettext("pg_pattern type is for %s is "
2998                                    "%d bytes too long\n"), service->sc_name,
2999                                    out_len - max_scf_name_len);
3000                        }
3001                }
3002        } else {
3003                const char *prefix;

3005                /* Make sure that the name is valid. */
3006                if (uu_check_name((const char *)name, UU_NAME_DOMAIN) != 0) {
3007                        semerr(gettext("pg_pattern name attribute, \"%s\", "
3008                            "for %s is invalid\n"), name, service->sc_name);
3009                        goto out;
3010                }

3012                /*
3013                 * As long as the pg_pattern has a name, it becomes part of
```



```
3014                 * the name of the pg_pattern property group name.  We
3015                 * merely need to pick the appropriate prefix.
3016                 */
3017                if ((type == NULL) || (*type == 0)) {
3018                        prefix = SCF_PG_TM_PG_PATTERN_N_PREFIX;
3019                } else {
3020                        prefix = SCF_PG_TM_PG_PATTERN_NT_PREFIX;
3021                }
3022                if ((out_len = snprintf(pg_name, max_scf_name_len + 1, "%s%s",
3023                    prefix, name)) >= max_scf_name_len + 1) {
3024                        uu_die(gettext("pg_pattern property group name "
3025                            "for %s is %d bytes too long\n"), service->sc_name,
3026                            out_len - max_scf_name_len);
3027                }
3028        }

3030        /*
3031         * Create the property group for holding this pg_pattern
3032         * information, and capture the pg_pattern attributes.
3033         */
3034        pg = internal_pgroup_create_strict(service, pg_name,
3035            SCF_GROUP_TEMPLATE_PG_PATTERN);
3036        if (pg == NULL) {
3037                if ((name == NULL) || (*name == 0)) {
3038                        if ((type == NULL) ||(*type == 0)) {
3039                                semerr(gettext("pg_pattern with empty name and "
3040                                    "type is not unique in %s\n"),
3041                                    service->sc_name);
3042                        } else {
3043                                semerr(gettext("pg_pattern with empty name and "
3044                                    "type \"%s\" is not unique in %s\n"),
3045                                    type, service->sc_name);
3046                        }
3047                } else {
3048                        if ((type == NULL) || (*type == 0)) {
3049                                semerr(gettext("pg_pattern with name \"%s\" "
3050                                    "and empty type is not unique in %s\n"),
3051                                    name, service->sc_name);
3052                        } else {
3053                                semerr(gettext("pg_pattern with name \"%s\" "
3054                                    "and type \"%s\" is not unique in %s\n"),
3055                                    name, type, service->sc_name);
3056                        }
3057                }
3058                goto out;
3059        }

3061        /*
3062         * Get the pg_pattern attributes from the manifest and verify
3063         * that they satisfy our restrictions.
3064         */
3065        r = lxml_get_pg_pattern_attributes(pg, pg_pattern);
3066        if (r != 0)
3067                goto out;
3068        if (verify_pg_pattern_attributes(service, pg) != 0) {
3069                semerr(gettext("Invalid pg_pattern attributes in %s\n"),
3070                    service->sc_name);
3071                r = -1;
3072                goto out;
3073        }

3075        /*
3076         * Now process all of the elements of pg_pattern.
3077         */
3078        for (cursor = pg_pattern->xmlChildrenNode;
3079            cursor != NULL;
```

```
3080             cursor = cursor->next) {
3081                 if (lxml_ignorable_block(cursor))
3082                         continue;

3084                 switch (lxml_xlate_element(cursor->name)) {
3085                 case SC_COMMON_NAME:
3086                         (void) lxml_get_all_loctext(service, pg, cursor,
3087                             COMMON_NAME_FMT, (const char *)cursor->name);
3088                         break;
3089                 case SC_DESCRIPTION:
3090                         (void) lxml_get_all_loctext(service, pg, cursor,
3091                             DESCRIPTION_FMT, (const char *)cursor->name);
3092                         break;
3093                 case SC_PROP_PATTERN:
3094                         r = lxml_get_tm_prop_pattern(service, cursor,
3095                             pg_name);
3096                         if (r != 0)
3097                                 goto out;
3098                         break;
3099                 default:
3100                         uu_die(gettext("illegal element \"%s\" in pg_pattern "
3101                             "for service \"%s\"\n"), cursor->name,
3102                             service->sc_name);
3103                 }
3104         }

3106 out:
3107         if ((r != 0) && (pg != NULL)) {
3108                 internal_detach_pgroup(service, pg);
3109                 internal_pgroup_free(pg);
3110         }
3111         free(pg_name);
3112         xmlFree(name);
3113         xmlFree(type);

3115         return (r);
3116 }

3118 static int
3119 lxml_get_template(entity_t *service, xmlNodePtr templ)
3120 {
3121         xmlNodePtr cursor;

3123         for (cursor = templ->xmlChildrenNode; cursor != NULL;
3124             cursor = cursor->next) {
3125                 if (lxml_ignorable_block(cursor))
3126                         continue;

3128                 switch (lxml_xlate_element(cursor->name)) {
3129                 case SC_COMMON_NAME:
3130                         (void) lxml_get_tm_common_name(service, cursor);
3131                         break;
3132                 case SC_DESCRIPTION:
3133                         (void) lxml_get_tm_description(service, cursor);
3134                         break;
3135                 case SC_DOCUMENTATION:
3136                         (void) lxml_get_tm_documentation(service, cursor);
3137                         break;
3138                 case SC_PG_PATTERN:
3139                         if (lxml_get_tm_pg_pattern(service, cursor) != 0)
3140                                 return (-1);
3141                         break;
3142                 default:
3143                         uu_die(gettext("illegal element \"%s\" on template "
3144                             "for service \"%s\"\n"),
3145                             cursor->name, service->sc_name);
```



```
3146                 }
3147         }

3149         return (0);
3150 }

3152 static int
3153 lxml_get_default_instance(entity_t *service, xmlNodePtr definst)
3154 {
3155         entity_t *i;
3156         xmlChar *enabled;
3157         pgroup_t *pg;
3158         property_t *p;
3159         char *package;
3160         uint64_t enabled_val = 0;

3162         i = internal_instance_new("default");

3164         if ((enabled = xmlGetProp(definst, (xmlChar *)enabled_attr)) != NULL) {
3165                 enabled_val = (strcmp(true, (const char *)enabled) == 0) ?
3166                     1 : 0;
3167                 xmlFree(enabled);
3168         }

3170         /*
3171          * New general property group with enabled boolean property set.
3172          */

3174         i->sc_op = service->sc_op;
3175         pg = internal_pgroup_new();
3176         (void) internal_attach_pgroup(i, pg);

3178         pg->sc_pgroup_name = (char *)scf_pg_general;
3179         pg->sc_pgroup_type = (char *)scf_group_framework;
3180         pg->sc_pgroup_flags = 0;

3182         p = internal_property_create(SCF_PROPERTY_ENABLED, SCF_TYPE_BOOLEAN, 1,
3183             enabled_val);

3185         (void) internal_attach_property(pg, p);

3187         /*
3188          * Add general/package property if PKGINST is set.
3189          */
3190         if ((package = getenv("PKGINST")) != NULL) {
3191                 p = internal_property_create(SCF_PROPERTY_PACKAGE,
3192                     SCF_TYPE_ASTRING, 1, package);

3194                 (void) internal_attach_property(pg, p);
3195         }

3197         return (internal_attach_entity(service, i));
3198 }

3200 /*
3201  * Translate an instance element into an internal property tree, added to
3202  * service.  If op is SVCCFG_OP_APPLY (i.e., apply a profile), set the
3203  * enabled property to override.
3204  *
3205  * If op is SVCCFG_OP_APPLY (i.e., apply a profile), do not allow for
3206  * modification of template data.
3207  */
3208 static int
3209 lxml_get_instance(entity_t *service, xmlNodePtr inst, bundle_type_t bt,
3210     svccfg_op_t op)
3211 {
```

```
3212         entity_t *i;
3213         pgroup_t *pg;
3214         property_t *p;
3215         xmlNodePtr cursor;
3216         xmlChar *enabled;
3217         int r, e_val;

3219         /*
3220          * Fetch its attributes, as appropriate.
3221          */
3222         i = internal_instance_new((char *)xmlGetProp(inst,
3223             (xmlChar *)name_attr));

3225         /*
3226          * Note that this must be done before walking the children so that
3227          * sc_fmri is set in case we enter lxml_get_dependent().
3228          */
3229         r = internal_attach_entity(service, i);
3230         if (r != 0)
3231                 return (r);

3233         i->sc_op = op;
3234         enabled = xmlGetProp(inst, (xmlChar *)enabled_attr);

3236         if (enabled == NULL) {
3237                 if (bt == SVCCFG_MANIFEST) {
3238                         semerr(gettext("Instance \"%s\" missing attribute "
3239                             "\"%s\".\n"), i->sc_name, enabled_attr);
3240                         return (-1);
3241                 }
3242         } else {        /* enabled != NULL */
3243                 if (strcmp(true, (const char *)enabled) != 0 &&
3244                     strcmp(false, (const char *)enabled) != 0) {
3245                         xmlFree(enabled);
3246                         semerr(gettext("Invalid enabled value\n"));
3247                         return (-1);
3248                 }
3249                 pg = internal_pgroup_new();
3250                 (void) internal_attach_pgroup(i, pg);

3252                 pg->sc_pgroup_name = (char *)scf_pg_general;
3253                 pg->sc_pgroup_type = (char *)scf_group_framework;
3254                 pg->sc_pgroup_flags = 0;

3256                 e_val = (strcmp(true, (const char *)enabled) == 0);
3257                 p = internal_property_create(SCF_PROPERTY_ENABLED,
3258                     SCF_TYPE_BOOLEAN, 1, (uint64_t)e_val);

3260                 p->sc_property_override = (op == SVCCFG_OP_APPLY);

3262                 (void) internal_attach_property(pg, p);

3264                 xmlFree(enabled);
3265         }

3267         /*
3268          * Walk its child elements, as appropriate.
3269          */
3270         for (cursor = inst->xmlChildrenNode; cursor != NULL;
3271             cursor = cursor->next) {
3272                 if (lxml_ignorable_block(cursor))
3273                         continue;

3275                 switch (lxml_xlate_element(cursor->name)) {
3276                 case SC_RESTARTER:
3277                         (void) lxml_get_restarter(i, cursor);
```



```
3278                         break;
3279                 case SC_DEPENDENCY:
3280                         (void) lxml_get_dependency(i, cursor);
3281                         break;
3282                 case SC_DEPENDENT:
3283                         (void) lxml_get_dependent(i, cursor);
3284                         break;
3285                 case SC_METHOD_CONTEXT:
3286                         (void) lxml_get_entity_method_context(i, cursor);
3287                         break;
3288                 case SC_EXEC_METHOD:
3289                         (void) lxml_get_exec_method(i, cursor);
3290                         break;
3291                 case SC_PROPERTY_GROUP:
3292                         (void) lxml_get_pgroup(i, cursor);
3293                         break;
3294                 case SC_TEMPLATE:
3295                         if (op == SVCCFG_OP_APPLY) {
3296                                 semerr(gettext("Template data for \"%s\" may "
3297                                     "not be modified in a profile.\n"),
3298                                     i->sc_name);

3300                                 return (-1);
3301                         }

3303                         if (lxml_get_template(i, cursor) != 0)
3304                                 return (-1);
3305                         break;
3306                 case SC_NOTIFICATION_PARAMETERS:
3307                         if (lxml_get_notification_parameters(i, cursor) != 0)
3308                                 return (-1);
3309                         break;
3310                 default:
3311                         uu_die(gettext(
3312                             "illegal element \"%s\" on instance \"%s\"\n"),
3313                             cursor->name, i->sc_name);
3314                         break;
3315                 }
3316         }

3318         return (0);
3319 }

3321 /* ARGSUSED1 */
3322 static int
3323 lxml_get_single_instance(entity_t *entity, xmlNodePtr si)
3324 {
3325         pgroup_t *pg;
3326         property_t *p;
3327         int r;

3329         pg = internal_pgroup_find_or_create(entity, (char *)scf_pg_general,
3330             (char *)scf_group_framework);

3332         p = internal_property_create(SCF_PROPERTY_SINGLE_INSTANCE,
3333             SCF_TYPE_BOOLEAN, 1, (uint64_t)1);

3335         r = internal_attach_property(pg, p);
3336         if (r != 0) {
3337                 internal_property_free(p);
3338                 return (-1);
3339         }

3341         return (0);
3342 }
```

```
3344 /*
3345  * Check to see if the service should allow the upgrade
3346  * process to handle adding of the manifestfiles linkage.
3347  *
3348  * If the service exists and does not have a manifestfiles
3349  * property group then the upgrade process should handle
3350  * the service.
3351  *
3352  * If the service doesn't exist or the service exists
3353  * and has a manifestfiles property group then the import
3354  * process can handle the manifestfiles property group
3355  * work.
3356  *
3357  * This prevents potential cleanup of unaccounted for instances
3358  * in early manifest import due to upgrade process needing
3359  * information that has not yet been supplied by manifests
3360  * that are still located in the /var/svc manifests directory.
3361  */
3362 static int
3363 lxml_check_upgrade(const char *service) {
3364         scf_handle_t    *h = NULL;
3365         scf_scope_t     *sc = NULL;
3366         scf_service_t   *svc = NULL;
3367         scf_propertygroup_t     *pg = NULL;
3368         int rc = SCF_FAILED;

3370         if ((h = scf_handle_create(SCF_VERSION)) == NULL ||
3371             (sc = scf_scope_create(h)) == NULL ||
3372             (svc = scf_service_create(h)) == NULL ||
3373             (pg = scf_pg_create(h)) == NULL)
3374                 goto out;

3376         if (scf_handle_bind(h) != 0)
3377                 goto out;

3379         if (scf_handle_get_scope(h, SCF_FMRI_LOCAL_SCOPE, sc) == -1)
3380                 goto out;

3382         if (scf_scope_get_service(sc, service, svc) != SCF_SUCCESS) {
3383                 if (scf_error() == SCF_ERROR_NOT_FOUND)
3384                         rc = SCF_SUCCESS;

3386                 goto out;
3387         }

3389         if (scf_service_get_pg(svc, SCF_PG_MANIFESTFILES, pg) != SCF_SUCCESS)
3390                 goto out;

3392         rc = SCF_SUCCESS;
3393 out:
3394         scf_pg_destroy(pg);
3395         scf_service_destroy(svc);
3396         scf_scope_destroy(sc);
3397         scf_handle_destroy(h);

3399         return (rc);
3400 }

3402 /*
3403  * Translate a service element into an internal instance/property tree, added
3404  * to bundle.
3405  *
3406  * If op is SVCCFG_OP_APPLY (i.e., apply a profile), do not allow for
3407  * modification of template data.
3408  */
3409 static int
```



```
3410 lxml_get_service(bundle_t *bundle, xmlNodePtr svc, svccfg_op_t op)
3411 {
3412         pgroup_t *pg;
3413         property_t *p;
3414         entity_t *s;
3415         xmlNodePtr cursor;
3416         xmlChar *type;
3417         xmlChar *version;
3418         int e;

3420         /*
3421          * Fetch attributes, as appropriate.
3422          */
3423         s = internal_service_new((char *)xmlGetProp(svc,
3424             (xmlChar *)name_attr));

3426         version = xmlGetProp(svc, (xmlChar *)version_attr);
3427         s->sc_u.sc_service.sc_service_version = atol((const char *)version);
3428         xmlFree(version);

3430         type = xmlGetProp(svc, (xmlChar *)type_attr);
3431         s->sc_u.sc_service.sc_service_type = lxml_xlate_service_type(type);
3432         xmlFree(type);

3434         /*
3435          * Set the global missing type to false before processing the service
3436          */
3437         est->sc_miss_type = B_FALSE;
3438         s->sc_op = op;

3440         /*
3441          * Now that the service is created create the manifest
3442          * property group and add the property value of the service.
3443          */
3444         if (lxml_check_upgrade(s->sc_name) == SCF_SUCCESS &&
3445             svc->doc->name != NULL &&
3446             bundle->sc_bundle_type == SVCCFG_MANIFEST) {
3447                 char *buf, *base, *fname, *bname;
3448                 size_t  base_sz = 0;

3450                 /*
3451                  * Must remove the PKG_INSTALL_ROOT, point to the correct
3452                  * directory after install
3453                  */
3454                 bname = uu_zalloc(PATH_MAX + 1);
3455                 if (realpath(svc->doc->name, bname) == NULL) {
3456                         uu_die(gettext("Unable to create the real path of the "
3457                             "manifest file \"%s\" : %d\n"), svc->doc->name,
3458                             errno);
3459                 }

3461                 base = getenv("PKG_INSTALL_ROOT");
3462                 if (base != NULL && strncmp(bname, base, strlen(base)) == 0) {
3463                         base_sz = strlen(base);
3464                 }
3465                 fname = safe_strdup(bname + base_sz);

3467                 uu_free(bname);
3468                 buf = mhash_filename_to_propname(svc->doc->name, B_FALSE);

3470                 pg = internal_pgroup_create_strict(s, SCF_PG_MANIFESTFILES,
3471                     SCF_GROUP_FRAMEWORK);

3473                 if (pg == NULL) {
3474                         uu_die(gettext("Property group for prop_pattern, "
3475                             "\"%s\", already exists in %s\n"),
```

```
3476                            SCF_PG_MANIFESTFILES, s->sc_name);
3477                }
3479                p = internal_property_create(buf, SCF_TYPE_ASTRING, 1, fname);
3481                (void) internal_attach_property(pg, p);
3482        }
3484        /*
3485         * Walk its child elements, as appropriate.
3486         */
3487        for (cursor = svc->xmlChildrenNode; cursor != NULL;
3488            cursor = cursor->next) {
3489                if (lxml_ignorable_block(cursor))
3490                        continue;
3492                e = lxml_xlate_element(cursor->name);
3494                switch (e) {
3495                case SC_INSTANCE:
3496                        if (lxml_get_instance(s, cursor,
3497                            bundle->sc_bundle_type, op) != 0)
3498                                return (-1);
3499                        break;
3500                case SC_TEMPLATE:
3501                        if (op == SVCCFG_OP_APPLY) {
3502                                semerr(gettext("Template data for \"%s\" may "
3503                                    "not be modified in a profile.\n"),
3504                                    s->sc_name);
3506                                return (-1);
3507                        }
3509                        if (lxml_get_template(s, cursor) != 0)
3510                                return (-1);
3511                        break;
3512                case SC_NOTIFICATION_PARAMETERS:
3513                        if (lxml_get_notification_parameters(s, cursor) != 0)
3514                                return (-1);
3515                        break;
3516                case SC_STABILITY:
3517                        (void) lxml_get_entity_stability(s, cursor);
3518                        break;
3519                case SC_DEPENDENCY:
3520                        (void) lxml_get_dependency(s, cursor);
3521                        break;
3522                case SC_DEPENDENT:
3523                        (void) lxml_get_dependent(s, cursor);
3524                        break;
3525                case SC_RESTARTER:
3526                        (void) lxml_get_restarter(s, cursor);
3527                        break;
3528                case SC_EXEC_METHOD:
3529                        (void) lxml_get_exec_method(s, cursor);
3530                        break;
3531                case SC_METHOD_CONTEXT:
3532                        (void) lxml_get_entity_method_context(s, cursor);
3533                        break;
3534                case SC_PROPERTY_GROUP:
3535                        (void) lxml_get_pgroup(s, cursor);
3536                        break;
3537                case SC_INSTANCE_CREATE_DEFAULT:
3538                        (void) lxml_get_default_instance(s, cursor);
3539                        break;
3540                case SC_INSTANCE_SINGLE:
3541                        (void) lxml_get_single_instance(s, cursor);
```



```
3542                        break;
3543                default:
3544                        uu_die(gettext(
3545                            "illegal element \"%s\" on service \"%s\"\n"),
3546                            cursor->name, s->sc_name);
3547                        break;
3548                }
3549        }
3551        /*
3552         * Now that the service has been processed set the missing type
3553         * for the service.  So that only the services with missing
3554         * types are processed.
3555         */
3556        s->sc_miss_type = est->sc_miss_type;
3557        if (est->sc_miss_type)
3558                est->sc_miss_type = B_FALSE;
3560        return (internal_attach_service(bundle, s));
3561 }
3563 #ifdef DEBUG
3564 void
3565 lxml_dump(int g, xmlNodePtr p)
3566 {
3567        if (p && p->name) {
3568                (void) printf("%d %s\n", g, p->name);
3570                for (p = p->xmlChildrenNode; p != NULL; p = p->next)
3571                        lxml_dump(g + 1, p);
3572        }
3573 }
3574 #endif /* DEBUG */
3576 static int
3577 lxml_is_known_dtd(const xmlChar *dtdname)
3578 {
3579        if (dtdname == NULL ||
3580            strcmp(MANIFEST_DTD_PATH, (const char *)dtdname) != 0)
3581                return (0);
3583        return (1);
3584 }
3586 static int
3587 lxml_get_bundle(bundle_t *bundle, bundle_type_t bundle_type,
3588     xmlNodePtr subbundle, svccfg_op_t op)
3589 {
3590        xmlNodePtr cursor;
3591        xmlChar *type;
3592        int e;
3594        /*
3595         * 1.  Get bundle attributes.
3596         */
3597        type = xmlGetProp(subbundle, (xmlChar *)type_attr);
3598        bundle->sc_bundle_type = lxml_xlate_bundle_type(type);
3599        if (bundle->sc_bundle_type != bundle_type &&
3600            bundle_type != SVCCFG_UNKNOWN_BUNDLE) {
3601                semerr(gettext("included bundle of different type.\n"));
3602                return (-1);
3603        }
3605        xmlFree(type);
3607        switch (op) {
```

```
3608         case SVCCFG_OP_IMPORT:
3609                 if (bundle->sc_bundle_type != SVCCFG_MANIFEST) {
3610                         semerr(gettext("document is not a manifest.\n"));
3611                         return (-1);
3612                 }
3613                 break;
3614         case SVCCFG_OP_APPLY:
3615                 if (bundle->sc_bundle_type != SVCCFG_PROFILE) {
3616                         semerr(gettext("document is not a profile.\n"));
3617                         return (-1);
3618                 }
3619                 break;
3620         case SVCCFG_OP_RESTORE:
3621                 if (bundle->sc_bundle_type != SVCCFG_ARCHIVE) {
3622                         semerr(gettext("document is not an archive.\n"));
3623                         return (-1);
3624                 }
3625                 break;
3626         }

3628         if (((bundle->sc_bundle_name = xmlGetProp(subbundle,
3629             (xmlChar *)name_attr)) == NULL) || (*bundle->sc_bundle_name == 0)) {
3630                 semerr(gettext("service bundle lacks name attribute\n"));
3631                 return (-1);
3632         }

3634         /*
3635          * 2.  Get services, descend into each one and build state.
3636          */
3637         for (cursor = subbundle->xmlChildrenNode; cursor != NULL;
3638             cursor = cursor->next) {
3639                 if (lxml_ignorable_block(cursor))
3640                         continue;

3642                 e = lxml_xlate_element(cursor->name);

3644                 switch (e) {
3645                 case SC_XI_INCLUDE:
3646                         continue;

3648                 case SC_SERVICE_BUNDLE:
3649                         if (lxml_get_bundle(bundle, bundle_type, cursor, op))
3650                                 return (-1);
3651                         break;
3652                 case SC_SERVICE:
3653                         if (lxml_get_service(bundle, cursor, op) != 0)
3654                                 return (-1);
3655                         break;
3656                 }
3657         }

3659         return (0);
3660 }

3662 /*
3663  * Load an XML tree from filename and translate it into an internal service
3664  * tree bundle.  Require that the bundle be of appropriate type for the
3665  * operation: archive for RESTORE, manifest for IMPORT, profile for APPLY.
3666  */
3667 int
3668 lxml_get_bundle_file(bundle_t *bundle, const char *filename, svccfg_op_t op)
3669 {
3670         xmlDocPtr document;
3671         xmlNodePtr cursor;
3672         xmlDtdPtr dtd = NULL;
3673         xmlValidCtxtPtr vcp;
```



```
3674         boolean_t do_validate;
3675         char *dtdpath = NULL;
3676         int r;

3678         /*
3679          * Verify we can read the file before we try to parse it.
3680          */
3681         if (access(filename, R_OK | F_OK) == -1) {
3682                 semerr(gettext("unable to open file: %s\n"), strerror(errno));
3683                 return (-1);
3684         }

3686         /*
3687          * Until libxml2 addresses DTD-based validation with XInclude, we don't
3688          * validate service profiles (i.e. the apply path).
3689          */
3690         do_validate = (op != SVCCFG_OP_APPLY) &&
3691             (getenv("SVCCFG_NOVALIDATE") == NULL);
3692         if (do_validate)
3693                 dtdpath = getenv("SVCCFG_DTD");

3695         if (dtdpath != NULL)
3696                 xmlLoadExtDtdDefaultValue = 0;

3698         if ((document = xmlReadFile(filename, NULL, 0)) == NULL) {
3699                 semerr(gettext("couldn't parse document\n"));
3700                 return (-1);
3701         }

3703         document->name = safe_strdup(filename);
1562         document->name = strdup(filename);

3705         /*
3706          * Verify that this is a document type we understand.
3707          */
3708         if ((dtd = xmlGetIntSubset(document)) == NULL) {
3709                 semerr(gettext("document has no DTD\n"));
3710                 return (-1);
3711         } else if (dtdpath == NULL && !do_validate) {
3712                 /*
3713                  * If apply then setup so that some validation
3714                  * for specific elements can be done.
3715                  */
3716                 dtdpath = (char *)document->intSubset->SystemID;
3717         }

3719         if (!lxml_is_known_dtd(dtd->SystemID)) {
3720                 semerr(gettext("document DTD unknown; not service bundle?\n"));
3721                 return (-1);
3722         }

3724         if ((cursor = xmlDocGetRootElement(document)) == NULL) {
3725                 semerr(gettext("document is empty\n"));
3726                 xmlFreeDoc(document);
3727                 return (-1);
3728         }

3730         if (xmlStrcmp(cursor->name, (const xmlChar *)"service_bundle") != 0) {
3731                 semerr(gettext("document is not a service bundle\n"));
3732                 xmlFreeDoc(document);
3733                 return (-1);
3734         }


3737         if (dtdpath != NULL) {
3738                 dtd = xmlParseDTD(NULL, (xmlChar *)dtdpath);
```

```
3739                 if (dtd == NULL) {
3740                         semerr(gettext("Could not parse DTD \"%s\".\n"),
3741                             dtdpath);
3742                         return (-1);
3743                 }

3745                 if (document->extSubset != NULL)
3746                         xmlFreeDtd(document->extSubset);

3748                 document->extSubset = dtd;
3749         }

3751         if (xmlXIncludeProcessFlags(document, XML_PARSE_XINCLUDE) == -1) {
3752                 semerr(gettext("couldn't handle XInclude statements "
3753                     "in document\n"));
3754                 return (-1);
3755         }

3757         if (do_validate) {
3758                 vcp = xmlNewValidCtxt();
3759                 if (vcp == NULL)
3760                         uu_die(gettext("could not allocate memory"));
3761                 vcp->warning = xmlParserValidityWarning;
3762                 vcp->error = xmlParserValidityError;

3764                 r = xmlValidateDocument(vcp, document);

3766                 xmlFreeValidCtxt(vcp);

3768                 if (r == 0) {
3769                         semerr(gettext("Document is not valid.\n"));
3770                         xmlFreeDoc(document);
3771                         return (-1);
3772                 }
3773         }

3775 #ifdef DEBUG
3776          lxml_dump(0, cursor);
3777 #endif /* DEBUG */

3779         r = lxml_get_bundle(bundle, SVCCFG_UNKNOWN_BUNDLE, cursor, op);

3781         xmlFreeDoc(document);

3783         return (r);
3784 }
```

_____***unchanged_portion_omitted_***