Tracking Specification Requirements Evolution: Database Approach

Denis Silakov
Institute for System Programming
of the Russian Academy of Sciences
Moscow, Russian Federation
Email: silakov@ispras.ru

***Abstract*—This paper presents a novel approach of tracking requirements history based on the usage of so-called specification database. The approach was applied to the Linux Standard Base specification. Both necessary database schema elements and required modifications of tools dealing with the database are concerned.**

***Index Terms*—Data management, Software requirements and specifications, Software standards.**

I. Introduction

In the modern software world standards and software specifications are of great importance. By uniforming sophisticated software products they allow developers to write programs that can interact successfully with each other without studying different details of the interaction target.

A common way to create software standards texts is to simply write them manually with the deep study of subjects in question and carefull checking of the text. Usually different templates are used to simplify development process, but nevertheless the part of manual efforts is usually huge. And the common way to show dependencies between different specification items is to add appropriate references to the specification document (which can be hyperlinks to referenced elements if document format is used that supports such features). But even with such references it is quite difficult to obtain a full view of sophisticated item interactions and interconnections which can be required during software development.

In order to create software product meeting some standard, developers have to study the text of this standard. Most standards are available in the form of printed books or electronic documents only. They usually contain a lot of pages and require a lot of time for human to study them in detail. In many cases development process could be simplified if some aspects of the standard, such as items interconnections mentioned above, are available in some other form, more formal than manually written text. For example, if one wants his program to use only operating system API (Application Programming Interface) specified by some standard and located in fixed header files, it would be useful to have such header files containing only API functions specified in the standard. If a program can be successfully compiled using only these header files, then one can guarantee that it doesn't require any external functions, which are not specified in the standard. But manual creation of such files is very time-consuming (for example, generic part of Linux Standard Base (LSB) 3.1 specification [1] contains description of 31425 functions from 409 headers accompanied by more than 6088 constants and 3964 types required for their proper declaration [2]).

Very often (especially when developing small applications) programmers prefer to rely on some tests performed after the program is completed or at least on their knowledge of the specification. But if a forbidden interface usage is discovered after the program has been created, then its elimination will cost much more than if it is caught directly after first appearance in the source code.

Moreover, standards are evolving and very often it is necessary for developers to know difference between standard versions (for example, in order to estimate how difficult it will be to support the new version if the software already supports the previous one). For the most standards such differences are published as well as texts themselves, but these are also text files which should be studied manually.

It is likely that standard creators have necessary information in a more convenient form, but most standards are created by special committees and the creation process is private inside these committees. There is a set of specifications whose text is available online at no cost (for example, POSIX), but they are not accompanied by any additional files; all what one can get except specification text is a test suite aimed to check applications compliance.

Some approaches exist aimed to extract necessary data from the textual specification and to upload it into the database – see, for example, [3]. But such approaches are not very useful without support from the specification developers since they require a lot of efforts and in many ways duplicate specification developers work. Therefore it is necessary to have specification developing approach allowing not only to create text itself, but simplifying different specification versions maintenance (i.e. evolution history of the specification elements and their interconnections) and different associated files creation.

The remainder of the paper is structured as follows. Section 2 describes an existing approach to develop specification text and associated files based on so-called specification database usage and applied by the Linux Foundation in the process of Linux Standard Base specification development. Section 3 introduces extension of this approach that allows one to

keep full history of the specification and to generate items concerning the certain version with minimal efforts. Section 4 describes the application of this extended approach to the LSB development process. Finally, Section 5 summarizes the main ideas.

II. Specification Database

A very promising approach is used by the Linux Foundation [4] for the Linux Standard Base (LSB) specification development. The main objective of the LSB is to unify ABI (Application Binary Interface) of different Linux distributions. Existence of such uniform ABI is demanded by software vendors (especially by those who have products developed for other operating systems and want to port them on Linux). Among the main problems of software development for Linux many vendors mention the need of recompiling their programs for every Linux distribution. Existence of the uniform ABI allows applications which use only this ABI to be executed on any distributions meeting LSB requirements without recompilation.

The main part of the LSB is a list of binary interfaces which should present in distribution. As for detailed interfaces descriptions, for most interfaces the specification references other existing standards (such as POSIX, System V ABI, etc.). Only interfaces that are not described in other documentation are specified directly by the LSB (the last version of the LSB, 3.1 Update 1, contains specifications for 458 interfaces, with the total amount of interfaces been equal to 36102).

The specification text of the LSB is mainly enumerations of interfaces grouped by header files they are declared in and accompanied by necessary types and constants declarations. In addition, a link to the document where the interface behavior is described in detail is provided (if it is not the LSB itself).

But the Linux Foundation's objective is not only to write specification text unifying different Linux distributions, but to provide distribution and application vendors with a lot of useful tools simplifying development of the LSB compliant software. That's why they need something more powerful than simple textual data.

Linux Standard Base is being developed using a database approach with a lot of data concerning the specification been stored in so-called *specification database*. Specification database greatly simplifies creation of both specification text itself and different files associated with it.

For each kind of elements specified by the LSB a separate table exists in the database containing different element properties. The full list of such tables includes:

- Module – modules are the top-level LSB entities, grouping all other elements according to their functionality (for example, LSB_Cpp module contains all C++ relative aspects, LSB_Qt3 module describes all entities concerning Qt3 library, etc.);
- Library – shared libraries that should provide interfaces specified by the LSB; from binary application point of view library is simply a set of interfaces;
- Command – except binary interfaces, the LSB also specifies some commands that should exist in distribution and can be used by applications. Commands include both shell built in commands and system utilities;
- ClassInfo – different information about classes for the C++ part of the specification;
- Header – although LSB is a binary standard, it is useful for developers to know in which header file each interface is declared (but this is not obligatory, and if a certain distribution contains interface definition in a header file differing from one mentioned in the LSB, this fact doesn't break LSB compliance);
- Interface – the main entity of the LSB; binary interfaces can be of the following kinds:
  - Function;
  - Data;
  - Common (such as stdin, stdout, etc.);
  - Alias (interfaces that are aliases of other interfaces);
- Type – this table stores different types of interface parameters with all base types; types are also grouped by headers;
- Constant – in order to make sure that interface declared in any header will work properly, it is sometimes necessary to ensure that some constants declared in header files have certain values;
- ElfSections, DynamicEntries – different aspects concerning Executable and Linking Format;
- RpmTags – description of RPM file sections.

The database also contains information about interconnections between elements (e.g. each constant and type is assigned to header it is declared in, each interface is assigned to library it is provided by and to class if it is a class member, etc.). In addition, interfaces and classes are grouped in so-called *LibGroups*; types and constant are grouped in *HeaderSections*. Header Sections have an ordering which allows to print declaration of one sets of types and constants before others (which scan be necessary for complex type declarations).

Entity-Relationship diagram of the existing database is shown on Fig.1. The diagram shows only tables corresponding to elements specified by the LSB. Many-to-Many relationships are implemented by auxiliary tables where different specific properties can also be held. Some tables are accompanied by additional tables for convenience purposes (for example, there is *TypeMember* table where members of complex types, such as structures, unions or enumerations are stored, there is a set of tables storing information about class virtual tables, etc.).

Since Linux Standard Base is a binary specification, it is architecture specific. Seven architectures are supported by the LSB at the moment (IA32, IA64, AMD64, PPC32, PPC64, S390 and S390X) and for each of these architectures a separate LSB specification document exists. There is also so-called *generic* specification, describing aspects that should be the same on all architectures.

In order to handle architecture specific questions, *Architecture* table exists in the specification database containing

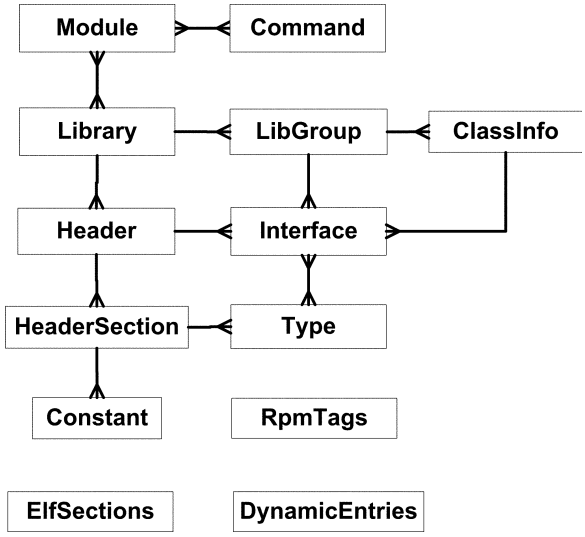

Fig. 1. LSB Specification Database Entity-Relationship Diagram

supported architectures list. There are also tables implementing Many-to-Many relationships between element tables and the Architecture table. Such tables (named ArchLib, ArchClass, ArchType, etc.) specify on which architecture each element is valid and contain architecture specific element properties (such as type size, constant value, binary interface version, etc.).

Detailed description of all LSB specification database tables can be found at the wiki portal of the LSB Infrastructure Project [5]. Since the LSB is a standard from the Open Source World, all things related to it including the database and different generators dealing with the database can be freely obtained from the Linux Foundation Bazaar repository. Detailed instructions for downloading and setting up all database relative stuff can be also found at the portal.

III. Specification Versions Maintenance

For a large variety of software standards and specifications the only way to obtain specification text or other files for previous specification versions is to restore the appropriate file versions from the repository (such as CVS or other version control system). And to study the specification history in this case is to explore differences between different file versions. But the existence of the database allows one to keep not only elements included in the specification, but the full history of this specification, i.e. to keep track of all changes between different specification versions with minimal efforts.

Historical aspects one might want to track can vary for different specifications, but for almost all of them the following historical data could be necessary for each element defined:

- Specification versions where the element was included;
- Specification versions where the element was mandatory (or those where the element was optional);
- Specification versions where the element became deprecated;
- The element's properties changes during specification evolution.

As for database tables where fields containing history information can be located, we can mention the following:

- If specification database already contains an appropriate table for each kind of elements we want to track history for, then necessary fields can be simply added to these tables (which particular fields might be required will be discussed below).
- If there are tables where different element properties are stored and we want to keep track of these properties, too, then it can be useful to add necessary fields there.
- In any case, one could create a separate table for each kind of element and place historical data there. Elements in these tables must have unique identifiers, or the other way should exist to determine which element this history information belongs to.

Different combinations of these variants are possible, and for different elements one can choose different fields locations. But one should keep in mind that it is desirable to have an unique way of storing history information for all kinds of elements; after all, we are trying to keep history of a standard that unifies something, and it's better to have some internal standard for internal infrastructure organization in order to simplify this infrastructure support.

There are also lots of other possibilities where to place historical information – for example, it is possible to have not one historical record per element, but compress historical information and to introduce one record per many elements. However, such extensions will hardly reduce database size considerably, but will add unwarranted complexity to the database schema, complicating database support and introducing a lot of place of potential inconsistencies.

Real-life database schemas can be very complicated and decisions where to place historical data should be make for each particular database, but one should keep in mind these general prudential aspects. Application of our approach to the LSB specification database described in Section 3 might be very useful in this way, since the database schema is complicated enough and various problems were faced there.

An important observation concerns element properties changes. Such changes can be caused both by evolution or by errors made in previous specification versions (although all software standards are strictly verified, they can nevertheless contain errors). We stongly recommend to treat both these changes as changes that should be tracked, though temptation can arise to simply correct the mistake found. The thing is that software products certified for the specification version with a mistake may not pass certification process for specification where this mistake is corrected. Such situations should be avoided.

In what follows we assume that we have a specification database and for each kind of elements we want to track history for there is an appropriate table containing different information about these elements. As it was mentioned above, if separate tables exist storing different properties we want to track, then such fields should be added to these tables.

Moreover, for some kind of elements it can be necessary to keep history information in more than one table. For example, let's assume that we have interfaces which are on the one hand assigned to header files and on the other hand assigned to some logical groupings (let's call them *Modules*). Let's also suppose that information about interfaces, headers and modules is stored in Interface, Header and Module tables respectively, relationship between Interfaces and Headers is 1-to-N relationship and is implemented by 'Iheader' field in the Interface table (pointing to the header identifier in the Header table), and relationship between interfaces and modules is Many-to-Many relationship implemented by ModuleInterface table, containing pairs *(InterfaceId; ModuleId)*. As specification evolves, interface can move from one header to another and from one module to other one. These movements can be independent and we'd like to track both kinds of migrations. Moreover, interface can be included or excluded from specification not depending on its header or modules, and this information is also of great interest. In order to store all needed information in this case we suggest the following database extensions:

- Add new fields to the Interface table indicating specification versions where interfaces was included, excluded, optional, etc. We will discuss below which fields can be required in details.
- Add new fields to the ModuleInterface table indicating in which specification versions interface was assigned to module.
- Create new table in order to track interface migration between header and drop 'Iheader' field. The new table will contain information about header to which interface was assigned in every version it was included.

The last point is less elegant than the previous ones since it introduces additional complexity to the schema and leads to more sophisticated requests. As an alternative we can suggest not to create new table, but to insert new record in the Interface table when interface is moved from one header to another. In this case the old record is marked as excluded from the new specification version, and new record is marked as included one. If interface migrations are rare, then this technique is much more preferable.

Let's now proceed with database schema extensions required to track historical aspects we have pointed out. Among possible ways of organizing tracking of such features we have regarded the following ones:

- Use simple text fields for each element containing lists of separated specification versions where the element was included, list of versions where it was optional and where it was deprecated. As a separator any symbol can be chosen which never appears in the specification version names. Let's call such fields 'IncludedIn', 'OptionalIn' and 'DeprecatedIn'.
- Use fields that will only indicate versions where an element was included for the first time, where it was withdrawn for the first time, till which version it was optional (or in which version it became mandatory) and in which version it became deprecated (i.e. track only versions where the element status was changed). Let's call such fields 'AppearedIn', 'WithdrawnIn', 'OptionalTill' (or 'MandatorySince') and 'DeprecatedSince'.

Note that in the first case there is no need in field containing list of versions where element was excluded from the specification – it is quite reasonable to suggest that if an element was not included, then it was excluded.

Versions to be stored in fields in both methods are not obligatory specification versions, since the latter ones usually can contain not only numbers, but whole words and even sentences. In many aspects it would be better to have a separate table containing all versions with unique identifiers assigned to them. These identifiers can be used as version fields values, saving space occupied by the database and introducing explicit ordering among versions (it is not always easy to say which of two versions in question preceeds another, and in any way number comparison is executed faster then string comparison – the fact important for the second method, as we will see later).

The other important observation is that elements can be included in some specification versions, be withdrawn in some successive versions and then be included again. All such changes also should be handled and different techniques can be suggested in order to track them. If the first technique mentioned above is followed, then such modifications are actually tracked automatically, but in the second case some additional tricks are required.

A. Tracking All Versions

When using this method, quite simple queries are required to obtain simple historical information. In order to get records from 'Element' table corresponding to elements that were included in '1.0' version, one should perform the following query:

SELECT * FROM Element WHERE IncludedIn like '%1.0%'

Although this query is very simple, one should remember that it can be quite slow if the 'IncludedIn' field is not indexed for full text search – simple index is not enough in this case. With every new version the average size of such fields will grow, index size will grow, too, and at some moment size of the database part storing history information can become quite big. Using identifiers instead of version names can decrease database size a little, but even in this case 'like' operator should be used, and although full text index exists, the performance will not be better than one when comparing two numbers.

It also should be noticed that this technique is not very suitable for manual data manipulation, since it is quite hard for human to explore long text strings looking for specific symbols and tools automating this manipulation process are required.

One more very important observation is that no version identifier should be a substring of another one, otherwise our

queries will provide us with data we don't actually want. For example, if we have two versions marked '1.0' and '1.0.1' and will perform the query given above, we will obtain all elements included either in '1.0' or in '1.0.1' or in both of them. So if one uses numbers as version identifiers, then it is better to use the numbers of the same length (i.e. containing the same amount of digits). And if one uses strings, it should be checked manually that no substrings present.

The last disadvantage of this method we'd like to mention is that of necessity of total database upgrade when new specification version is coming. Even if there is no changes concerning a given element, a new version identifier should be appended to its 'IncludedIn' field. To be sure, this process can be automated by many ways. For example, a new version identifier can be first automatically appended to 'IncludedIn' fields of all elements included in the previous version, and then eliminated from fields of elements which should be withdrawn. Alternatively, a list of interfaces that should be included in new version can be prepared in advance and then all data can be set automatically according to this list. Which method is more preferable depends on different particular aspects of the development process, on kind of differences between new and old versions, etc. However, any technique introduces considerable overhead which is absent when the second approach is used.

B. Tracking Versions with Status Changes Only

For this method SQL queries can become more complex even for requests with simple semantics, depending on some additional conditions. First, let's assume that versions are represented by numbers. Other thing which is also important is default values for 'AppearedIn' and 'WithdrawnIn' fields (i.e. for cases when the element has never been added or withdrawn from the specification). If we use zero values by default, then in order to get all records from the Element table corresponding to entities which were included in the specification with version '1', one has to perform the following query:

SELECT * FROM Element WHERE AppearedIn $>$ 0 AND AppearedIn $<=$ 1 AND (WithdrawnIn $=$ 0 OR WithdrawnIn $>$ 1)

One can mention that one of the possible ways of simplifying these queries is to use some very large numbers as default values (infinity values will be perfect), but such solution may seem to be not very elegant for users exploring database contents manually. Another possible simplification comes from the fact that for most specifications there is no need to store elements that have never been included, so the presense of zero 'AppearedIn' field can be regarded as inconsistency and there is no need to compare it to zero in query given above.

Another problem already mentioned above is that one of needs for some tricks in order to store history for elements that have been added to specification and withdrawn from it not once. First, let's consider 'AppearedIn' and 'WithdrawnIn' fields only. In order to track history changes using these two fields, we can suggest the following techniques:

- Provide availability of storing more than one record for each element by adding 'AppearedIn' field to element tables primary keys;
- Add another one field, say 'WasExcludedIn'. In this case the 'AppearedIn' field should contain specification version where the element appeared for the first time, the 'WithdrawnIn' field should contain version where the element was withdrawn for the last time, and the new "WasExcludedIn" field should contain intermediate versions (between "AppearedIn" and "WithdrawnIn" values) where the element was excluded.

Introduction of a new 'WasExcludedIn' field simply turns our database schema into some kind of one used when all version are stored where an element were included and discussed in the previous section. So one will have to use 'like' operators in order to know if an element was included in some version, missing one of the main advantage of the method in question.

The former solution is more preferebale, since it is quite clear for users and introduces no extra complexity to the database schema. If it is not usual for a given specification to move large sets of elements from included to excluded status and back again very often, then such approach will not enlarge the database considerably.

The same solutions can be applied for other fields, such as 'OptionalTill' or 'DeprecatedSince'. The former solution seems to be even more preferable here if we take a look at the semantics of these fields. For example, usually between versions where element was added and where it was withdrawn it can become deprecated only once, so one will not have to create any extra records.

C. Consistency Questions

When one has some data stored in the database and interconnections exist between different parts of this data it is desirable to have the data and all interconnections consistent. This can be reached either by means of the database management system used or by some external tools checking database data for consistency (if database management system methods are unavailable or harder to implement or undesired for some other reasons).

First, agreement should be met about optional and deprecated elements. It is a common way that if an element is deprecated or optional (or mandatory), then this element is included in the specification (i.e. excluded elements can not have any additional status, such as deprecated or optional). In what follows we will follow this way, too.

Major consistency points concerning historical data vary for two techniques described in previous sections. If all versions are stored in a text filed, as described in Section 3.A, then we can point out the following consistency aspects:

- If a version identifier is included in 'OptionalIn' or 'DeprecatedIn' field for some element, then this version must present in 'IncludedIn' field, too. Surely, the agreement can be made that if an element is optional or deprecated, then it is automatically included (even if it is not so

according to the 'IncludedIn' field), but in this case one will not be able to check if an element was marked optional or deprecated by accident. So it's just a kind of reinsurance.
- Each specification version identifier should appear in each field not more than once. Although this is not a major issue, it can lead to inconveniences during data exploration.

For the second technique (described in Section 3.B) Let's assume for simplicity that a field *is empty* if it doesn't contain any actual version identifier (which value will be used in pratice for these purposes doesn't matetr here), otherwise let's say that a field *contains some value*. For this technique more consistency checks can be pointed out:

- If 'WithdrawnIn' field contains some value then the 'AppearedIn' field for this element must not be empty and its value must be less than 'WithdrawnIn'.
- If 'OptionalTill' or 'DeprecatedSince' fields contain some values, then 'ApperedIn' field must have value lesser then their ones.
- If 'WithdrawnIn' field is not empty and 'OptionalTill' or 'DeprecatedSince' fields also contain some values, then these values must be less than 'WithdrawnIn'.
- If 'MandatorySince' field is used instead of 'OptionalTill' and if it contains some value then this value should also be greater or equal to 'AppearedIn' and if 'WithdrawnIn' is not empty then 'MandatorySince' must be lesser than 'WithdrawnIn'.
- For each element there should not exist different records in the same tables whose ('AppearedIn'..'WithdrawnIn') intervals intersect (if 'WithdrawnIn' field is empty, then it is convenient here to handle it as it is equal to infinity). I.e. if there is a record saying that some element was added in '1.0' and withdrawn in '3.0', then there should be no record saying that this element was added in '2.0'. The thing is that such two records might contain different properties for the same element and it is not clear which of them were included in '2.0' version.

An interesting problem arises from the fact that history information for the same element can be stored in different tables. In example from the beginning of Section 3 data for interfaces is stored at least in two tables – Interface and ModuleInterface. From human point of view, an interface can be assigned to a module independently of its specification status, i.e. even excluded interfaces can be assigned to some modules. But we recommend to use the uniform way of storing history information in all tables, otherwise misunderstanding can arise which especially undesirable when writing automatic tools dealing with the database. Therefore in the example mentioned we recommend to mark interface as excluded from module (i.e. set appropriate mark in the ModuleInterface table) if it is excluded from the specification itself. Thus, one more consistency requirement arises:

- History information for the same element located in different tables must be consistent, i.e. if element is excluded in '1.0' version according to some table, it should be marked as excluded in '1.0' version in all tables.

All consistency checks mentioned in this section are rather simple and each of them can be performed using one SQL query. Nevertheless, they are very important, since if any of them fail for some elements, then one can conclude that actually it is impossible to say anything particular about evolution of these elements.

IV. Keeping Track of the LSB Versions

The approach suggested in the previous section has been successfully applied in the joint project of ISP RAS and Linus Foundation dedicated to the LSB Infrastructure Project. The LSB specification database described in Section 2 was improved in order to track information about different LSB versions. The aim was not only to have historical data in the database and to allow performing historical queries, but to adjust all file generators used by the Linux Foundation that deal with the specification database in order to generate data corresponding to a given LSB version.

Availability of tools for a given LSB version can be very useful in some cases. For example, different distributions are certified on different LSB versions. And if a software vendor wants to check quickly his application against different LSB versions without official certification process (simply to know on which distributions his program can be executed successfully without recompilation) he may use tools provided with the LSB. The only thing he needs in order to get such tools for every LSB version is a set of scripts generating them and a local copy of the database.

Most of the elements defined in the LSB can be architecture specific – they may have different properties on different architectures and the same element can be included in the LSB only for some of the architectures supported. That's why all fields containing history information for architecture specific elements were placed into tables implementing Many-to-Many relationship between *Architecture* table and elements' ones. More particular, new fields were added to the following tables:

- Module – for modules. which are the same on all architectures.
- ArchLib – for libraries.
- ModLib – also for libraries, since the same library can be assigned to different modules in different specification versions.
- ModCmd – for commands, since the same commands are specified for all architectures, but the same command may be assigned to different modules in different specification versions. It should be noticed here that new fields automatically track command's status in the specification and there is no need in fields in any other places (e.g. in the Command table).
- ArchClass – for classes.
- Header – for header files; header files themselves are not architecture specific, only their contents does (i.e.

TABLE I
RECORDS IN THE ARCHDE TABLE FOR DYNAMIC ENTRY INCLUDED TWICE

ADEdename	ADEappearedin	ADEwithdrawnin
DT_FLAGS_1	1.0	1.2
DT_FLAGS_1	1.3	2.0

constants, types or interface declarations can be enclosed in the appropriate *ifdef* directives).
- ArchInt – for interfaces.
- ArchConst – for constants.
- ArchType – for types.
- TypeMember – for members of complex types, such as structures, unions or enumerations. Type members can be architecture specific, but all architecture information is stored in the TypeMember table, too. Such decision was made in order to avoid too complex requests.
- ArchDE – for dynamic entries.
- ArchES – for elf sections.
- RpmTag – for rpm tags, which are architecture independent.

We have chosen the second technique described in Section 3, i.e. we have decided to add fields of 'AppearedIn' and 'WithdrawnIn' kind. (Actual fields were named according to the LSB specification database naming conventions – 'ACappearedin' and 'ACwithdrawnin' for the ArchClass table, AESappearedin and AESwithdrawnin for the ArchES table, etc.).

According to the approach, for elements that were included in the LSB more than once, additional records were inserted in the appropriate tables. For example, dynamic entry DT_FLAGS_1 was first added in the LSB 1.0, withdrawn in the LSB 1.2, added once again in the LSB 1.3 and finally dropped in the LSB 2.0. For this entry, there are two records in the ArchDE table, as shown in the Table I.

Accordind to the Linux Foundation, the only element that can be optional in the LSB is module. For tracking optional modules history a new field, 'Mmandatorysince', was added to the Module table. As for deprecated interfaces, there are no examples of them yet in the LSB and there is no clear strategy yet in making elements deprecated. But as soon as need in marking elements as deprecated will arise and clear agreement will be met on how to do this from the specification's point of view, the appropriate database schema changes will be performed. According to the approach used, new fields of 'DeprecatedSince' kind will be added to some tables (but it is not clear now to which ones, since it is not clear if an element can be deprecated on one architecture and not to be deprecated on the others).

When the LSB specification database schema upgrade was performed, the LSB already had a quite long history which was not reflected in the database. All necessary data for earlier LSB versions was restored semi-automatically on the basis of the LSB CVS repository where database snapshots are stored (in a textual form) for every specification version. The amount of data was huge and scripts written in order to automate restoration process were also programs and could not be guaranteed to be free of errors. Therefore results obtained using these scripts were verified carefully, and here different scripts checking data consistency as mentioned in Section 3.C had a lot of use. The final way to check if the data is correct was to generate specification text for every version and compare it to one that was officially published and also available in the LSB repository, but generation process takes a lot of time (several hours), while the average execution time of the consistency checkers was about several minutes and these checkers helped to find about 90% of errors made during data restoration.

Some tools provided with the LSB use files generated on the basis of the specification database data. These tools are:

- So called *Build Environment* aimed to help developers to write LSB compatible software. If a program can be compiled in this environment, that one can guarantee that it is LSB compliant [6]. The environment includes:
  - Stub libraries – libraries containing only interfaces specified in the LSB. These interfaces are not real interfaces, but dummy stubs, i.e. they do not perfrom any useful operations. The only aim of these libraries is to ensure that application in question doesn't require any other interfaces.
  - Header files – the headers containing declarations of only those interfaces that are specified by the LSB. They also contain types, constants and macro declarations in order to make interface declarations valid. For some interfaces and macro to be declared properly some other interfaces must be declared in the header. Such interfaces are also stored in the specification database with special marks indicating that interface should be included in header files only.
- Different checkers that can be used in order to find rough inconsistencies between the specification and realization:
  - cmdchk – checks that all commands specified by the LSB present in the system and that they are located in the proper directories, if it is required by the specification.
  - devchk – checks types and constants declarations in system header files against information stored in the database.
  - dynchk – dynamic checking of application's usage of the interfaces. For some interaces general restrictions exist (such as *the first parameter must not be NULL*) which are also stored in the database. This tool checks if application violates these restrictions during execution.
  - elfchk – static tests for executable files in order to ensure that they require only interfaces allowed by the LSB.
  - libchk – checks all shared libraries required by the LSB with full contents (i.e. checks if they provide all binary interfaces required by the specification).

– rpmchk – checks rpm files for the LSB compliance.

Some files for these tools are generated by scripts using SQL queries in order to obtain elements with their properties included in the specification. More detailed information about these checkers generation using the database can be found in [7]. Information about build environment header files and stub libraries generation is presented in [8].

All the scripts and queries inside them were improved in order to be able to generate files corresponding to a given LSB version using queries introduced in the Section 3.B.

The other very useful application of the historical data is providing availability of different queries for historical data that allow one to simply obtain different statistics in a short time. Exploring a separated versions of the database corresponding to different specification versions would require much more time and efforts. In order to simplify browsing different information about elements stored in the LSB specification database, ISP RAS is developing a web interface available to visualize different elements. their and interconnections. This web interface is called LSB DB Navigator and available online at the site of the Linux Foundation [9]. After introducing version support in the specification database availability of browsing elements with respect to a certain LSB version was also added to the LSB DB Navigator. Moreover, different statistics pages are available presenting number of interfaces, commands and headers for each LSB version on every architecture.

LSB specification database is continually evolving and more different entities related to the LSB will be stored in this database in future and for some of these entities evolution tracking may also be desirable. For example, now database extensions are being developed that should provide availability of storing information about test suites and coverage they provide. In these extensions history specific aspects will also be in use, indicating which sets of interfaces were covered by test suites in different LSB versions.

V. Conclusion

This paper has suggested an approach of tracking requirements history of software specifications and standards based on the database usage. The method solves the problem of generation of files corresponding to a given specification version on demand. The files include both specification text itself and different tools associated with the specification and simplifying specification usage in practice. The approach also greatly simplifies specification evolution statistics gathering problem.

According to the approach described, the existed specification database of the Linux Standard Base was extended in order to store evolutional information. Historical data was added for all elements described by the LSB. All tools dealing with the database were improved in order to be able to generate files corresponding to a given LSB version and a convenient web interface was developed in order to explore different historical data and statistics about the LSB evolution.

Historical data for previous LSB versions (released before the database improvement) was gathered on the basis of corresponding database snapshots. In order to verify the data uploaded, consistency checks suggested by the approach were used which allowed to discover about 90% of incorrectly uploaded data in a short time.

The approach can be used not only for software standards, but for any standards specifiying large sets of interconnected elements. In order to apply the approach, standard itself should meet the following requirements:

- Elements described by standard should be divided on clear unambiguous groups (such as interfaces, libraries and types in software standards). For each such group a separate database table will be provided to track evolution information.
- There should be a certain set of possible states for each element in standard (e.g. 'Included', 'Excluded', 'Optional', 'Deprecated'). It is desirable (but not obligatory) for all kinds of elements to have the same possible state sets.

The approach suggested doesn't require the database with requirements to exist. Even in case when the specification is stored in the form of textual files, a simple database can be created containing only identifiers of all elements described by the specification and history information. It also should be mentioned here that the requirements database, when supported by the appropriate tools, helps not only to store evolutional information and to gather statistics, but to generate different files associated with the specification as well as specification text itself corresponding to a given specification version.

References

[1] Linux Standard Base. https://www.linux-foundation.org/en/LSB

[2] The LSB 3.1 Specification. http://refspecs.freestandards.org/LSB_3.1.0/

[3] Atsushi Ohnishi, "Software Requirements Definition for Specification Database," APSEC, p. 521, Fourth Asia-Pacific Software Engineering and International Computer Science Conference (APSEC'97 / ICSC'97), 1997.

[4] The Linux Foundation. https://www.linux-foundation.org

[5] LSB Infrastructure Project. http://ispras.freestandards.org

[6] Linux Standard Base Team. Building Applications with the Linux Standard Base. 2005. (http://lsbbook.gforge.freestandards.org/lsbbook.html)

[7] Stuart Anderson, Matt Elder. Run-time Testing of LSB Applications. Proceedings of the Linux Symposium, Volume 1, July 2004. (http://www.linuxsymposium.org/proceedings/reprints/Reprint-Anderson-OLS2004.pdf)

[8] Mats Whichmann. Increasing the Appeal of Open Source Projects. Proceedings of the Linux Symposium, Volume 2, July 2004. (http://www.linuxsymposium.org/proceedings/reprints/Reprint-Wichmann-OLS2004.pdf)

[9] LSB Database Navigator. https://www.linux-foundation.org/dbadmin/commons/welcome.php