Informational System to Support Development and Usage of Linux Interface Standards

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

***Abstract*—This paper presents an approach for developing Linux interface standards aimed to improve portability of applications among different Linux distributions. The approach is based on usage of database-driven informational system that simplifies creation and maintenance of interface standards by standardization committees and their usage by application and distribution developers. A logical model of interfaces between Linux applications and distributions is described which is used to design schema of the informational system's database.**

***Keywords*-Software requirements and specifications, Software standards, Data management.**

I. Introduction

The Linux operating system becomes more and more popular. Nowadays it is used not only by enthusiasts, but by many commercial companies, corporations and government organizations. Nevertheless, the market share of Linux in some areas (in particular, on desktops) is still relatively small. One of the main reasons which prevents the growth of Linux popularity in these market segments is lack of applications for this operating system that would satisfy all the needs of target audience.

This lack of applications arises, in particular, from a huge variety of existing operating systems based on the Linux kernel, GNU libraries and utilities and other common components. Such systems are called Linux *distributions*; there are several hundreds of distributions at the moment [1] and the situation is constantly changing – as time goes by, new distributions appear, while the others become obsolete and unsupported, but the total number of distributions is permanently increasing.

Most components that form a distribution are maintained not by distribution vendors themselves, but by different third party developers. This allows to save a lot of resources and efforts, but leads to another kind of problems. The thing is that many developers in the Open Source Software (OSS) world follow the "Release early, release often" policy [2], and it is not uncommon for software updates to appear several times a month. Such often releases lead to situations when a lot of different versions of the same component exist which, in general, provide different functionality. Moreover, distribution vendors often modify software taken from upstream, sometimes slightly, but sometimes significantly – for example, they can add some new unique functionality which will give more advantages to their system with respect to the others. As a result, functionality of the same component in different distributions can vary significantly.

A large variety of distributions provides users with a wide choice of Linux implementations, but such a variety makes it difficult to develop *portable* software that would be able to run in every Linux distribution without any additional actions from the user side. Approaches used by software vendors to increase the number of supported distributions depend on kind of license under which their programs are delivered. From licensing point of view, we should distinguish *open* software, whose source code can be obtained by interested parties for investigation and modification, and *closed*, or *proprietary* software, whose license forbids code modifications.

Developers of open source programs usually leave the task of software adoption for those distribution vendors who want to include their programs. In this case it is distribution engineers who test applications inside particular systems and modify their source code, if necessary. Finally, users themselves can build program from sources (and rely on programs like GNU Autotools that can take care of differences in build environments [3]).

Developers of proprietary software cannot follow this way. Instead, they have to provide binary executable files and shared libraries for their applications that are ready to use "as is", without recompilation or other actions. But it can be very expensive and time consuming to test some application in every existing distribution. That's why many proprietary vendors declare that they only support a few selected systems – usually those that have significant market share, such as SUSE Enterprise Linux or Red Hat Enterprise Linux (for example, IBM XL Fortran supports only these two distributions [4]; Intel Fortran Compiler supports seven systems [5], but this is also not a large number). However, end users normally expect to buy products "for Linux", not "for SUSE" or "for Red Hat".

A promising approach to simplify creation of portable applications distributions is *standardization* – development of requirements that should be satisfied by all standard compatible systems. In our case, *interface* standards are required guaranteeing that every compliant operating system provides certain interfaces (in particular, libraries and functions) that can be used by applications.

Standards are useful not only for proprietary vendors, but

also for developers of open source programs. The thing is that the more modifications are required to adopt a particular application for some distribution, the more likely the modified program will significantly differ from its origin and will be not exactly that thing which the original developer wants it to be. In addition, it's likely that if several programs exist providing the same functionality, then distribution vendors will choose those that require less efforts for maintenance and adoption. Following standards will give developers guarantees that their product will suite perfectly for any standard compliant system and will be unlikely subjected to significant modifications.

Modern Linux distributions are large and provide millions of interfaces of different kinds. For standardization committees, it is important to investigate which interfaces are mostly required and useful; due to a huge number of existing interfaces, some automation of this analysis is desired. But even with careful selection of standardized interfaces, standards can, in turn, become huge, so their size will cause problems for both standardization committees (responsible for standard maintenance and further development) and for developers, who will have to investigate thousands of pages of specification text. Thus, an approach is required to organize development process of an interface standard which will simplify both standard maintenance and development by appropriate committees and standard usage by its target audience – primarily, application and distribution developers.

The remainder of the paper is structured as follows: Section 2 observes the most valuable interface standards in the OSS world and analyzes approaches and techniques used during their development. Section 3 introduces an approach for interface standard development process organization which is based on using of database-driven informational system. Section 4 describes the application of the approach to the Linux Standard Base development process. Finally, Section 5 summarizes the main ideas.

II. Standards in the Open Source World

Portability problem is not a new one for Open Software, and standardization is declared to be one of the key principles of the Open Systems that should solve this problem (at least partially). However, even with such a principle, real life shows that it's not always easy to achieve full compatibility between different products. Problems arise in two areas – standard **development and maintenance** by standardization groups and committees and standard **usage** by its target audience – developers of applications and OS components.

Roots of the first problem lie in a huge number of existing libraries and functions – a modern Linux distribution delivered on a single DVD disk provides several hundreds of libraries which, in turn, export hundreds of thousands of functions. Not all of these functions can be considered as stable, safe, backward compatible, etc. – that is, not all functions can be characterized as a "best practice" and recommended to be used by everyone. One of the main tasks of standardization committee is to select those interfaces that are proved to be useful, and probably try to help to improve those interfaces which are not mature yet. That's why it is important to estimate real needs of applications, capabilities of existing Linux implementations and common practices used to solve particular problems, in order to standardize the mostly requested and important interfaces first. The more so, since besides such interface importance analysis, standardization process involves development and maintenance of specification text, tests and other accompanying products and informational resources – that is, standardization is actually an expensive and time-consuming task, so it is not desirable to waste resources.

Another effect of a large number of existing libraries and interfaces is that standards can become very large, too. This leads to the second problem – large specifications are hard to use for their target audience, since it's not easy to investigate a dozen volumes of specification, several hundreds of pages each. In order to make developers life easier, some standards are accompanied by auxiliary tools, informational resources and other additional products. A common example of such a product is a test suite that can be used to check if application meets all standard requirements. A more sophisticated example is a specialized development environment whose usage during the application compilation and build processes guarantees compliance of resulting program with the standard.

Such auxiliary components form a **standard environment**. All parts of this environment should be kept in sync with each other and with the specification text. For example, if it is decided to remove some interface from the specification, then the test suite for applications should be updated to forbid usage of this interface, the application development tools should be modified to avoid usage of this interface, and so on. Thus, while complicated and feature rich environment of a standard is useful for its target audience, it can significantly complicate development and maintenance of standard and accompanying tools.

One more issue of standardization we'd like to mention is that standards are not always fully suitable for every particular area. It's not uncommon when several standards exist that cover some area or when a small subset of a standard is enough for some class of systems. In such cases, standard **profiles** are developed – unions of existing standards or their subsets aimed to create a specification covering a certain class of systems. As for interface standardization, profiles are asked for when developing highly tailored products – for example, intended to be used only on high-loaded servers or inside mobile devices. Developers of such applications only consider operating systems that can work on their target platforms, and it would be useful for them to have a standard that describes only such particular class of systems. To be sure, existence of specifications that already cover (at least partially) target area can simplify development of a new document, and profile development is usually cheaper then development of a standard from scratch. However, it can introduce its own problems – when selecting subsets of existing standards and then joining these subsets into a single document, it is important to keep internal consistency of resulting specification.

In addition, it can be useful to reuse existing auxiliary tools, and these tools should be also adopted for a new profile – superfluous tests should be dropped, informational resources from different specifications that form the profile should be somehow combined and so on. Thus, profile development is not as cheap as it can seem to be.

All the problems mentioned above are not new and they were faced by different standardization workgroups. Let's consider different approaches used in order to solve them by some famous interface standards that are in use in the Linux world.

A. POSIX and SUS

The most famous and mature open standards for operating system interface are POSIX and Single UNIX Specification (SUS). Initially, these specifications were developed to achieve portability of applications among different UNIX implementations on the source level. This approach supposes standardization of the system Application Programming Interface (API), the core part of which are functions provided by system libraries and declared in appropriate header files. It is guaranteed that any application that meets requirements of some API standard can be compiled from its sources in any operating system compatible with that standard.

Roots of the Single UNIX Specification lie in the Common API Specification, developed in the early 1990th by the COSE alliance formed by all leading UNIX vendors of that time. The main purpose of this alliance was to investigate existing UNIX implementations and create a list of functions that were present in all UNIX systems. The resulting list contained 1170 functions and due to this reason it is also known as Spec 1170. In 1992-1993, during the SUS development, an additional research of 50 leading UNIX applications was performed and additional list of 130 functions was created that were suggested for standardization [7].

Application and distribution analysis during SUS and POSIX development was primarily performed manually and involved deep source code investigation by analysts. In early 1990th, this approach was suitable and allowed to perform a high quality and complete analysis.

A problem with initial versions of POSIX and SUS was that these standards considered only some relatively low level functions and calls, but this was not enough for many applications even in that time – such popular areas as graphical user interface or multimedia were completely out of standardization scope. The need for more areas was understood by standardization committees, and it was decided to develop several SUS profiles – specifications that were based on POSIX but extended it with interfaces specific to particular areas. The SUS version 2 specification presented three profiles – *Base Specification* (predecessor of POSIX 2001), *UNIX98 Workstation* (with GUI requirements based on the Common Desktop Environment – CDE – and the Motif library) and *UNIX98 Server* (specifying additional network services and Java Runtime Environment).

Unlike the base specification, extended profiles were suitable for UNIX-based systems only – for example, there were no free Motif and CDE implementations for Linux. Moreover, there were no concurrent implementations of CDE or Motif at all; concurrent implementations of some other standardized items were allowed, but they had to follow other existing specifications (like Java RE). Thus, during extended UNIX profile development, standardization workgroups didn't have to analyze alternative implementations, they only had to choose some top-level standardization directions – for example, once it was decided that CDE would be a standard desktop environment, there were no need to investigate different (and partially incompatible) implementations of CDE, since there was only one implementation of it in the wild.

On the other side, the POSIX itself was divided on several subsets that also formed a set of profiles – such as POSIX.1b real-time extensions. However, these profiles were even smaller than POSIX and their creation haven't require investigation of some new standardization techniques.

B. LSB

An alternative approach to API standardization is to standardize Application Binary Interface (ABI), giving developers an opportunity to use the same executable files and shared libraries in all compliant systems, without a need for recompilation. The core part of such ABI standards are shared libraries that should be provided by operating system and binary symbols exported by them (*binary symbol* is a binary level entity corresponding to either a function or a global variable exported by library). For application developers, this ABI standardization is more preferable than the one for API, since it doesn't require any actions (neither from developers nor from users) in order to port a program to any standard compliant system. However, ABI standards contain much more limitations for OS – in particular, it is clear that all target systems should use the same format for binary executables and shared libraries. That's why ABI standards often cover less systems than API ones.

Nowadays this approach is used by the Linux Standard Base specification (LSB) which is intended to be applied for Linux based systems only [12]. Roots of LSB lie in POSIX and SUS, and standardization process is also similar in many ways. In particular, LSB developers constantly perform analysis existing distributions and applications in order to select the mostly important and useful interfaces. Initially, the analysis process was also performed manually; but up to now the size of data that should be analyzed increased dramatically, and manual analysis doesn't work fine any more. In particular, during LSB 3.0 development, only interfaces provided by RHEL and SLES distributions were taken into account, while there were several hundreds of different Linux distributions in the world.

LSB has a rich environment, consisting of test suites, development environment for application vendors, online informational resources and other products. All these items are, on the one hand, independent products; on the other hand, they

all represent the LSB in some way and should be kept in consistency with it. The size of all these products makes it hard to perform such synchronization manually; in order to automate this task, a *specification database* was designed to store some information about standardized elements accompanied with a set of tools that were used to synchronize LSB environment components with each other and with LSB itself.

After LSB 3.0 was released and development of the next version was started, it became clear that the current infrastructure implies too many manual work and can't satisfy the all the needs of the LSB workgroup. In December, 2006, Ian Murdock (CIO of Free Standards Group that was responsible for LSB development at that moment) on the LSB Face-to-Face meeting formulated the following problems of the LSB Infrastructure [12]:

- absence of possibilities of Linux ecosystem analysis that would allow to effectively select further development directions;
- complexity of support of several LSB versions at once caused by absence of information about standard evolution in the database;
- high complexity of adding new interfaces to LSB – though the database solved the problem of synchronization of specification text and environment components, the task of populating database with data was not a trivial task;
- lack of auxiliary tools that would help distribution and application vendors to use LSB in the development process.

Summarizing POSIX and LSB experience, we can conclude that as the size of operating systems (measured in a number of interfaces) grows, the amount of work to be performed by standardization committees increases dramatically, and those approaches for standard development that proved to be useful a decade ago nowadays fail to satisfy all the needs of both standardization committees and those developers who use standards. New approaches are required that would help both standardization workgroups and standard users to perform their work effectively.

III. An Approach for Linux Interface Standards Development

In this paper, we present an approach for Linux Interface Standards Development. The approach includes the following stages:

1) Analysis of the Linux ecosystem:
   - selection of popular and mostly important applications, analysis of their requirements for system libraries and functions;
   - collection of information about existing distributions – in particular, about provided libraries and exported functions.

   The set of applications and distributions is constantly evolving, so it is necessary to have data not only with respect to some fixed time point, but collect information about the Linux ecosystem evolution during last several years. It is important to perform constant monitoring of the ecosystem, and results of this monitoring at some certain time points can be used to create next version of a standard, as demonstrated at Fig.1.

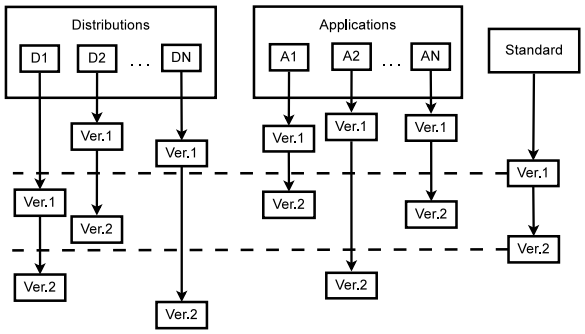

Fig. 1. Analysis of the Linux ecosystem during standard development

2) Preparation of a new standard version. This stage includes selection of interfaces which are mostly needed by applications, proved to be stable and provided by all modern distributions. Then, on the basis of this set of a consistent set of interfaces is constructed which will be included in the specification.
3) Addition of semantic information (in particular, descriptions of functionality that should be provided by interfaces), development of tests, adopting the standard certification system to support certification process for the new version and other tasks that should finalize release of a new standard version.

In order to support this method, we suggest to build an informational system which could be used to automate (at least partially) the mostly time-consuming tasks. The suggested informational system is based on a logical model of interfaces in the Linux ecosystem.

A. Logical Model of Application Interfaces with the Linux OS

In this paper, we concentrate on Application Binary Interface (ABI) – that is, we consider interfaces between binary executables and libraries of applications and shared libraries of distributions. Thus, we consider applications as a set of compiled files (executables and shared objects). In Linux, the main format used for such files is ELF ("Executable and Linking Format"). In our model, we'll include some items related to the ELF format; the general ELF description is provided by the System V ABI Specification [6]; some Linux specific extensions are described in the appropriate LSB sections [8].

All properties of any item which is a part of system ABI or API can be divided in two groups:

- **structural** properties, that can be checked statically – for example, names of functions exported by library or signature of any function from a given header file;

- **semantic** properties, whose analysis usually requires runtime testing – for example, function behavior.

The model described in this paper includes structural interface properties only, abstracting away from semantic aspects.

As elements represented in the model, we use interfaces involved in the process of **dynamic loading** of application files [10]. Compatibility between application and distribution with respect to such interfaces guarantees that application can be successfully *launched* in the distribution – that is, dynamic loader will be able to resolve all external dependencies of application, form the executable image in memory and pass the control to application's main entry.

The following interfaces are considered:

- **libraries** – a special kind of ELF files that can *export* interfaces;
- **binary symbols** exported by libraries – these are binary level entities corresponding to functions and global variables;
- **structure and size of types** used as function parameters and return values;
- **ELF file attributes** – *class* (32bit or 64bit), *target architecture* of a file and *types of sections* that exist in file.

Concentrating on application launching process, the model leaves out of account the following ways of interaction between Linux applications and distributions:

- dynamic loading of shared libraries and dynamic invocation of symbols exported by them at runtime (for example, using the *libdl* library capabilities);
- invocation of external commands and utilities at runtime (for example, using the *system* or *exec* functions).

However, modern recommendations on developing of portable applications forbid usage of such possibilities, unless all files involved in the interaction are part of the application. Indirect dependency on a system library or command cannot be checked by means of the operating system itself (e.g., by dynamic loader), so it is application developer who should check that necessary files exist and provide all required interfaces. However, such checks add complexity to any program, and improperly performed checks can lead to program crash or unexpected behavior [9].

B. Informational System to Support Development and Usage of Linux Interface Standards

In order to support the approach to interface standard development described above, we use an informational system providing the following possibilities:

- planning of further standard evolution;
- creation of new versions of standard and its profiles;
- ensuring consistency of standard environment components;
- checking of how different Linux distributions and applications are compliant with the standard.

The informational system is aimed to automate the most time consuming tasks that arise during the processes described above.

The main components of the system are like the following:

- a **database** with information about both standardized interfaces and interfaces used by existing applications and provided by distributions. The database schema is based on the logical model of interfaces described above;
- automated **data collection tools** used to gather information to populate the database with data;
- automated **generators** that use the database to create components of standard environment.

The database should store information about all interfaces with their characteristics described in the specification which are used by at least one component of the standard environment. If any component during its work requires some information about standardized interfaces which is described in the specification, this information should be either directly queried from the database when such a need occurs, or should be embedded in the component code at compilation time by appropriate automated generators. In particular, if some component needs to know the list of included interfaces, this list should be always taken from the database. This approach guarantees that all components are kept synchronized with each other and with specification text. To be sure, it is required for the specification text itself to be synchronized with the database; one of the ways to achieve it is to generate those parts of the text that are represented in the database – that is, the database should be the only one source of information about standardized items.

Besides the information about standardized items, the database should also contain all the data which is used by several components of standard environment, even though this data doesn't concern the standard itself. This will allow to keep different components synchronized with respect to their common data.

Due to a large number of interfaces that exist in the Linux world and should be subjected for analysis, data collection tools should be as automated as possible. Collection of data about interfaces included in our logical model can be almost fully automated, as demonstrated in authors' work [15]. Moreover, collection of additional information (e.g., header files) which is not used during Linux ecosystem analysis but required for development of different LSB environment components can be also automated significantly [14].

A data work flow diagram in out informational system is shown at the Fig.2.

In order to store information about interfaces that exist in the Linux ecosystem, we suggest to represent each kind of interface as two separate entities in the database schema – the first one will correspond to standardized interfaces of this kind, the other will represent interfaces which are present in distributions and used by applications. The reason is that information about standardized objects and data about ecosystem interfaces are used in different ways – the former is picked up by environment generators, the latter is supposed to be used during the ecosystem analysis, when planning further directions of standard development. In general, these

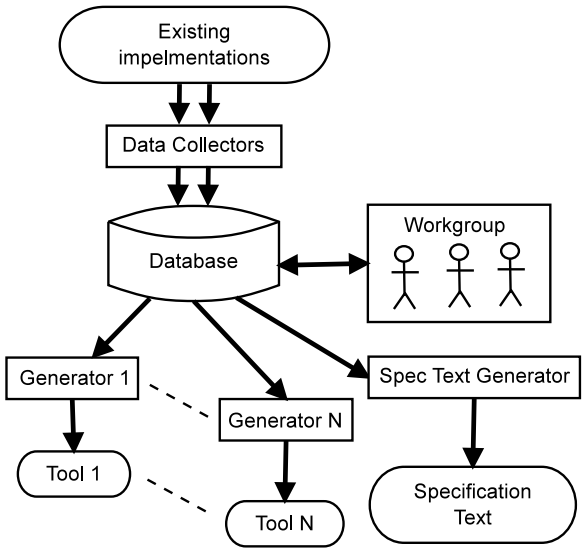

Fig. 2. Data work flow in the informational system

two tasks can require knowledge about different characteristics of the same interfaces. In particular, due to the big amount of existing interfaces that should be subjected to analysis, it can be reasonable to store only those ecosystem data that can be collected automatically; however, standards can be more descriptive and include more characteristics in addition to the collected ones, so the automatically collected data can be insufficient for environment generators.

In order to store information about several versions of a standard (that is, to store standard *history*), the database schema should be extended with attributes containing *temporal* data. Different approaches exist for introducing such extensions; in our work, we use the Temporal Relationship Model (TRM) [11], which is based on the relational model but adds new temporal attributes to every relation. With this model, there is no need to use a specialized temporal DBMS; the database can be served by any relational DBMS – the most popular and widespread kind of DBMS at the moment.

The two obligatory attributes added by the temporal model are the beginning and the end of entity *life period* – a time interval during which the entity preserves its characteristics. In our case, such interval boundaries are standard versions – that is, a time interval for some standardized item indicates a set of standard versions where this item was included with the same characteristics. A special value *NULL* is used to indicate unbounded intervals which correspond to items which exist in the last standard version (that is, that have been never excluded from the specification).

Temporal attributes are added only to those entities that correspond to standardized items; these attributes are not required for entities that represent interfaces existing in the Linux ecosystem. More details about using temporal databases for tracking standard evolution can be found in another author's work – [13].

IV. The LSB Infrastructure Program

One of the largest standards that specify interfaces of the Linux OS is Linux Standard Base (LSB). The standard is being developed by international consortium named The Linux Foundation which is formed by leaders of the Linux market. The primary content of the standard is formed by lists of libraries that should be present in any compliant Linux distribution, accompanied by lists of binary symbols that should be exported by these libraries. The standard is constantly evolving, and more and more interfaces are added – the latest version, LSB 4.0, describes more than 38.000 functions from 57 libraries. It is noticeable that during the four years passed from LSB 3.0 release, more than 30.000 functions were added.

Such a swift growth of the specification size exposed some significant problems in its development process and surrounding infrastructure. Among the most important issues, the lack of support for Linux ecosystem analysis was mentioned, as well as difficulties with specification text usage by application developers – even LSB 3.0 consisted of several thousands of pages and contained references to several dozens of other specifications [12].

In 2006, the joint Program of The Linux Foundation and Institute for System Programming of RAS was started aimed to improve the LSB Infrastructure. The main purpose of the Program was to resolve existent issues that made difficulties for standard maintenance; it was decided to create an informational system that would allow to both simplify further LSB development and simplify its usage by target audience – Linux application developers and distribution vendors.

By the beginning of the Program, the LSB infrastructure already contained a central database with information about standardized interfaces. That database was used to generate parts of the specification text (lists of libraries, binary symbols, etc.), to create header files and stub libraries for LSB Development Environment and to generate primitive tests checking presence of certain objects (libraries, commands, etc.) in distributions.

During the LSB Infrastructure Program, the following tasks were performed:

- an extension of the LSB database was developed called *Community Database* to store information about interfaces provided by existing Linux distributions and used by Linux applications; automated tools were developed to collect such data and populate the database with it. Nowadays that database contains information about 250 Linux distributions and 1200 applications;
- during the LSB Navigator development, automated tools were created aimed to support analysis of data about existing Linux distributions and applications during the LSB development process. The tools allow to discover potential candidates for standardization and check formal rules that should be met by candidates to be finally included in the specification;
- a temporal extension of the LSB database was developed

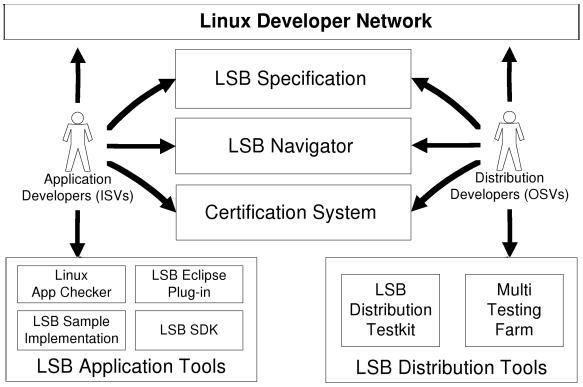

Fig. 3. LSB Environment structure

to store information about all existing LSB versions. All tools that use information from the database were modified to be able to extract data corresponding to any given specification version. Moreover, some products created using the database now support several LSB versions at once – in particular, the LSB Development Environment can be used to build applications compliant with any given LSB version.

Nowadays the work is in progress on improving profile support in the LSB Infrastructure, caused by a need to develop a profile for mobile devices.

The current structure of the LSB Environment is shown at the Fig.3.

The tools developed during the Program allowed to reorganize LSB development process – automation of many time-consuming tasks allowed LSB workgroup members to concentrate on their primary objective – selecting interfaces that should be included in the specification and elaborating descriptions of their behavior. Moreover, the decision making process itself was also significantly improved – the new infrastructure allowed to perform deeper analysis of the Linux ecosystem and to better understand current needs and evolution tendencies of applications and distributions. For example, during the LSB 3.0 development only two distributions were subjected to deep analysis (RHEL and SLES), and information about application needs was limited to direct requests from application developers (expressed in either LSB Bugzilla or mail lists). With the new infrastructure, during the LSB 4.0 development the workgroup analyzed all versions of 12 distributions released during the last three years and more than 1.000 applications.

This, in turn, allowed to significantly increase the number of standardized interfaces from 6.000 in LSB 3.0 to 38.000 in LSB 4.0. Nowadays we can say that the most significant problem with standardization of new interfaces is development of runtime tests; all other tasks (collecting data for the LSB database, keeping components of the LSB Development Environment synchronized, etc.) are highly automated and do not require much engineering efforts.

V. Conclusion

This paper has suggested and approach of developing Linux interface standards aimed to improve portability of applications among different Linux distributions. The approach is based on usage of a database-driven informational system that simplifies creation and maintenance of interface standards and their environment by standardization committees and their usage by application and distribution developers. A logical model of interfaces between Linux applications and distributions is described which is used to design schema of the informational system's database.

Usage of a central database to create different components of the standard environment allows to keep these components synchronized with each other and with the specification text automatically – every change in the database is automatically reflected in all components by means of appropriate generators. Temporal extensions of the database allow to store standard evolution history, which, in turn, allows to support several standard versions by means of the same database and accompanying tools.

Though in this paper we have considered ABI standards, the approach suggested is suitable for developing API standards, too. In order to support API specification, the model of interfaces between Linux applications and distributions should be modified – binary-only elements (e.g., ELF attributes) should be dropped, while entities that are present on source level only (e.g., constants and macros) should be added. Actually, the LSB database, described in this paper, already store some source-level entities and tools exist to automate collection of such information.

The LSB Infrastructure project has demonstrated the practical strength of the method of Linux interface standards development suggested in this paper. The informational system created during the project allowed to automate analysis of the Linux ecosystem and significantly increased the speed of decision making process. The automated data collection tools and database-driven generators eliminated the technical complexity of adding new interfaces to LSB. Finally, the new LSB Infrastructure supports development of profiles based on the LSB specification.

References

[1] The LWN.net Linux Distribution List. http://lwn.net/Distributions/
[2] E. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly Media, Inc.; Revised &; Expanded edition, 2001.
[3] G.V. Vaughan, B. Elliston, T. Tromey, I. Lance Taylor. *GNU Autoconf, Automake and Libtool*. Sams Publishing; online edition, 2006. [Online] Available: http://sources.redhat.com/autobook/
[4] IBM XL Fortran for Linux. [Online] Available: http://www-01.ibm.com/software/awdtools/fortran/xlfortran/linux/
[5] Intel Fortran Compiler Professional Edition 11.0 for Linux – Installation Guide and Release Notes. [Online] Available: http://cache-www.intel.com/cd/00/00/40/60/406087_406087.pdf
[6] System V Application Binary Interface. 24 April, 2001. [Online] Available: http://refspecs.linuxfoundation.org/elf/gabi4+/contents.html
[7] A. Josey. *API Standards for Open Systems.* The Open Group, 2001. [Online] Available: http://www.opengroup.org/austin/papers/wp-apis.txt

[8] Linux Standard Base Core Specication 4.0. Executable And Linking Format (ELF). [Online] Available: http://refspecs.linuxfoundation.org/LSB_4.0.0/LSB-Core-generic/LSB-Core-generic/elf-generic.html

[9] *Coding practices for compatibility.* Hewlett-Packard Developer & Solution Partner Program. [Online] Available: http://sysdoc.doors.ch/HP/compat.pdf

[10] M. Tim Jones. *Anatomy of Linux dynamic libraries.* IBM developerWorks, 2008. [Online] Available: http://www.ibm.com/developerworks/linux/library/l-dynamic-libraries/

[11] Abdullah Uz Tansel. "Temporal Relational Data Model." *IEEE Transactions on Knowledge and Data Engineering,* vol. 9, N3, pp. 464-479, May-June 1997.

[12] Ian Murdock. *LSB Overview and Progress Report.* LSB Face-to-Face Meeting. December 2006. [Online] Available: http://www.linuxfoundation.org/images/c/c2/Lsb-f2f-200612-overview.pdf

[13] D. Silakov. *Tracking Specification Requirements Evolution: Database Approach.* Proceedings of SYRCoSE 2007, vol. 2, pp. 15-22. Moscow, Russia.

[14] E. Novikov, D. Silakov. *The Automated Analysis of Header Files for Support of the Standardization Process.* Proceedings of SYRCoSE 2009, pp. 27-34. Moscow, Russia.

[15] D. Silakov. *Linux Distributions and Applications Analysis During Linux Standard Base Development.* Proceedings of SYRCoSE 2008, vol. 1, pp. 11-18. St.Petersburg, Russia.