Designing a Development Environment to Support Creation of Standard-Compliant Applications

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 of developing a special environment to help application developers to create programs compliant with some interface standard. The paper suggests a design of the informational system aimed to make it easier to develop and to maintain such an environment on the basis of the existing systems, with necessary modifications in the areas concerned by the standard. To store information about existing systems and to facilitate their modification, it is suggested to use a database with a set of accompanying tools. Necessary aspects of the database schema design are described, as well as some aspects of the tool architecture.**

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

I. Introduction

In the modern software world, it is a common situation when several concurrent implementations exist that provide the same functionality to their users (either human beings or other software applications). From the user's point of view, it would be nice if all systems with the same functionality (that are targeted to solve the same problems) have the same interaction interface – in this case it wouldn't be very hard to replace one implementation with another (that was found to be faster, more convenient, etc.) and users will not be bound to a particular solution. In order to achieve such interchangeability, different interface standards are being developed that specify the set of interfaces that every compliant system should provide.

However, in addition to the standard set of interfaces every system is usually not forbidden to provide some extra ones, that can be unique to it and absent on other platforms. This raises a problem for those developers who want to create applications interacting with the system by means of the standard interfaces only (in order to guarantee that the application will be able to interact with any other system compliant with the same standard). For example, in order to make the application sources be compilable by any C language compiler that is compliant with the ANSI C standard, developers should use only those language constructions and library routines that are defined in that standard; in order to make the application binaries able to be executed on any distribution compliant with the Linux Standard Base [1], they should use only LSB interfaces, and so on. But how to ensure that the application doesn't use any extra interfaces? To be sure, developers can consult documentation on every external interface used by the program. However, this can take significant time, especially if developers are not yet familiar with the standard. Moreover, sometimes dependencies on undesirable interfaces may appear indirectly, as a side effect of the environment used to build the program. For example, for binary executables some dependencies on binary interfaces can be introduced by compiler on the basis of its options, environment variables, etc. Sometimes it can be rather hard to detect and eliminate such indirect usage.

One of the possible ways to automate the control of used interfaces is to create a tool (or use an existing one, if any) that will check used interfaces and report any violation of the standard. If integrated in the build process as a part of the tests, such a tool can be used to detect wrong interfaces at early stages of the development process. However, the checker itself can only state a fact of violation, but not to point out the reason and help to fix it. In particular, it doesn't solve the problem of interfaces that are not used directly but introduced by the build environment.

An ideal solution would be to use a real system that is compliant with the standard and at the same time doesn't provide any non-standard interfaces. Unfortunately, for many standards such implementations don't exist, and it can be quite expensive (or even impossible) to create one from scratch. It appears to be more reasonable to take an existing system compliant with the standard, to cut its non-standard pieces and to set it up properly. Such a restricted environment can be either a self-sufficient system (like the LSB Sample Implementation [2]) or be integrated in the existing system, providing only those components that are covered by the standard (like the OpenGL Sample Implementation [3] or the LSB Development Environment [4]).

One should keep in mind here that every standard evolves over time, and all the tools accompanying the standard should be kept in sync with it. Furthermore, it is not uncommon when several versions of the same standard are demanded by software developers, so specification developers should either support a set of separate versions of helper tools or create tools that can support any specification version from some given set. Modern interface standards often describe hundreds or even thousands of interfaces and evolve quickly by adding more and more interfaces to satisfy market needs. Surely, all accompanying tools should not be left behind; however, the process of their development and support can be even harder then the development of the standard itself. Nowadays, an approach is required to organize, facilitate and automate (where possible) this process to allow specification creators to concentrate on their primary target – specification text.

The remainder of the paper is structured as follows: Section 2 observes some existing approaches of helping application developers to create standard-compliant programs. Section 3 introduces an approach that uses a database with information about standard elements to generate a desired environment in a semi-automatic way and allows to keep the environment in sync with the specification text and other relative tools. Section 4 describes the application of the approach to the LSB Development Environment creation process. Finally, Section 5 summarizes the main ideas.

II. Existing Approaches

One of the most popular approaches to check that the software product is compliant with some standard is to run appropriate tests (certification tests, if possible) against the final product. For example, if application developers target the Linux Standard Base specification, they can integrate the Linux Application Checker tool [5] to the testing process. When targeting Solaris OS, developers can use the appcert [6] tool to check if their binary files satisfy the rules defined in the Solaris ABI (e.g. don't use private symbols or don't link Solaris libraries statically).

Both advantages and disadvantages of this approach come from the fact that it doesn't affect the development process itself. The main advantage for the developers is that they should not make any modifications to their usual process; they should just add an additional test suite to be executed as a part of the product tests. Surely, this will increase the execution time of the tests, but this is usually not a great issue,

A real disadvantage of the approach is that it only allows to give a verdict on whether the product meets standard requirements. If the tests pass, then everything is alright; but in case of failures, the tests can only postulate the fact that a failure occurred, usually with description of the inconsistency found. It is application developer who should find the reason of the failure in the source code (or build environment) and fix it. Sometimes this may require just a little code modification, but in some cases this may lead to redesign of the product architecture. If no special actions to satisfy the standard were taken before the test execution, then nobody can predict how compliant the result product is and how much efforts will it take to make it fully compliant.

Any preliminary actions taken to make the product more compliant with the standard potentially make this product closer to the specification requirements, thus decreasing the probability of discovering serious inconsistencies when executing the tests. The most evident action is to consult the standard for every interface used by the program to check if it is allowed, which restrictions are put over it, which alternatives can be used, etc. This is probably reasonable in case when developers and architectures are familiar with the standard and can give the answer instantly, without actual addressing of the specification text. Thus, a certain level of expertise is required. Surely, the experience can be obtained through the practice, but for many standards (and for many developers) such straightforward practice as careful investigation of the specification text can significantly delay the release of the product.

A more complicated approach is to provide developers with a special environment that will simplify the process of achieving a standard-compliant product by discovering the inconsistencies as early as possible. For example, the environment can be constructed in such a manner that no program can be built inside it until all standard requirements are met.

The practice of providing such an environment is rather widespread in those areas where direct programming in the target system (where the application should be executed) is difficult – the system can be expensive and difficult of access for developers (e.g. operating systems on mainframes) or have limited resources making it impossible to run such tools as debugger or profiler (e.g. operating systems on mobile devices). In most of such cases, the ideal implementation (that is a system that provides those and only those interfaces that can be used by applications) does exist, but is hard to access. The common solution for this problem is to emulate the target environment on some other system which is more accessible for developers. In order to achieve this, one can either to use a hardware emulator to execute an existing system inside it, or recompile the system for another platform. The former case usually doesn't require any modifications of the system itself, but requires an appropriate hardware emulator. On the opposite, the latter one doesn't require any additional tools but its availability depends on adaptability of the system source code.

An example of the first approach is a QEMU-based Android Emulator [7] that provides a virtual ARM mobile device with a full Android system stack running on it. Android Emulator is an essential part of the Android SDK, which also includes a set of tools to interact with the emulator. An example of the second approach is the LSB Sample Implementation (SI) – a Linux distribution build on the basis of Linux From Scratch (LFS) [8] (until version 4.0) or rPath [9] (since 4.0). Both LFS and rPath allow to obtain a system meeting given requirements. However, none of these technologies is flexible enough to create a system with any given set of functions (moreover, it's not only tools fault – Linux components are sometimes interconnected very closely, and it can be impossible to cut one component without affecting the others). This fact complicates the development process of the LSB SI itself and it is actually not guaranteed that the result system provides no forbidden interfaces.

Another possible way of emulating a desired environment is to provide some mechanism for one of the existing systems to behave exactly as the desired one. Such approach can be used even if the desired environment doesn't exist in a pure form (i.e. if there is no real system that provides only allowed interfaces). However, required modifications of the system can be significant, and in case of complex systems it can be rather difficult to maintain these changes in parallel to the development of the main system. So this way is acceptable when implemented and maintained by developers of the main system, and when required possibility is an integral part of that system. As an example of this approach, we can mention C and C++ compilers from the GNU Compiler Collection that support '-std' option that can be used to determine the language standard to be followed by compiler (such as ISO C90 or ISO C++98).

The approaches mentioned above concern modifications and adaptation of rather complex software systems, such as operating system or compiler. Development of required modifications is usually a separate project, whose complexity depends on different conditions, such as size of required modifications, source code adaptability and so on. Sometimes the required efforts are minimalistic, sometimes they are not. In any case, at the current state of art a specific approach is used for every particular project of such kind, but almost all these projects imply manual examination of the system, detection of pieces that are not compatible with the given requirements and investigation of possible fixes for these pieces. Thus, such approaches are reasonable only if there is already a system that is rather close to the desired one.

However, in many cases there is no need to modify the whole huge software product (e.g. operating system). Instead, it is enough to provide alternative implementation for some of its parts. Note that there is a significant difference between providing an alternative implementation that coexists with the default one (or replaces the default one) and providing the whole system where the desired implementation is the only available one. The latter case requires a separate machine for the system to be executed (either a physical or virtual). In the former case, developers can use their usual systems in their usual work and use alternative environment during application development. In particular, this allows developers to use their habitual IDE, debugger and other tools that are hardly provided by the modified environment.

This way is actually the only reasonable approach for those standards that specify only a small part of the system interfaces. For example, the SGI's sample implementation of the OpenGL API [3] presents several hundreds of OpenGL functions. The main purpose of the OpenGL Sample Implementation is to give a standard base for other vendors of OpenGL implementations (in the first place, developers of implementations that use hardware acceleration), but it can be also used by application developers (since 1999, the Sample Implementation is an open source product) to ensure that they don't use any non-standard GL functions. However, it makes no sense to provide a separate operating system with this implementation, since such a system would have a very small difference with respect to any existing one. Moreover, such a system would not be very convenient for everyday usage, since the OpenGL SI doesn't include any hardware drivers and thus lacks for performance.

A similar way is followed by the LSB Development Environment, also known as the LSB Software Development Kit (LSB SDK), which is discussed in details in the Section 4. This product is designed for application developers and only provides headers, compiler wrapper and stub files for libraries to be used during the build process. The SDK can be used only to compile applications, but not to execute or test them. However, this is enough to guarantee that the application doesn't use non-LSB interfaces (except the cases when indirect calls such as one using the dlopen function are used). LSB SDK proved to be much more easier to be maintained than the LSB Sample Implementation, and on the opposite of the latter it guarantees that its header files and libraries don't provide forbidden interfaces. But since the SDK cannot be used to test applications, it is actually the chain of the SDK and SI that should be used in the development process.

This example is similar to the usage of cross-compilation to create executables for the hardware architectures that differ from the current one – in that case one needs a compiler that supports cross-compilation for the target platform, with appropriate header files and shared libraries. The whole operation system is not required for compilation, though some additional runtime environment is required to execute and test the application.

But even providing of only a subset of the complex system may require significant efforts on synchronizing the environment with the underlying software standard, especially if the standard evolves quickly. A promising approach of facilitating this task was introduced by the original creators of the LSB specification and its accompanying tools, including the LSB SDK – it was suggested to store those properties of the standardized items that are used by the tools in a central database and generate appropriate parts of the tools automatically using this database. Initially the LSB database contained names of standardized interfaces, their signatures and all types required to make these signatures complete. At the very beginning, the database was populated with data manually, but as LSB evolved, some tools were introduced to automate this process.

The rest of this paper summarizes and generalizes the approach used by the LSB team, giving general advises on how to automate the process of environment generation using the database, as well as appropriate data collection and manipulation processes. In addition, some further developments of this approach are given, including support of several versions of the standard by the same set of tools. In the Section 4, we'll return to the LSB SDK and describe its current state, with all improvements made by the authors.

III. Database-Driven Environment Generation

The approach under discussion suggests to pick out those parts of the tools to be created that are either specified by the standard or influenced by the standardized items. Information about such elements should be separated from data about other parts of the tools, that are not concerned by the standard in any way. That is, one should create a template of every tool – some kind of skeleton that just misses those parts that concern standardized items. These “holes” should be filled by additional *generator* on the basis of some external storage. In this paper, we consider a database to be such a storage, though the main ideas can be applied to any storage kind.

A. Database Design

First of all, one have to detect what should be stored in the database and design appropriate database schema. For every tool to be generated using the database, the following analysis should be performed:

- Determine entities involved in the tool that are defined in the standard.
- Determine entities that are influenced by the standardized items. This step is recursive – one should also determine entities influenced by those that are influenced by the standardize items, and so on, until a closed set is obtained.
- Determine which properties of entities picked out during the first two steps are necessary for the tool.

On the third step, it might be necessary to determine all properties required for the tool, even those that are not concerned by the standard. A clear rule can be used to determine if one should store all properties of some item in the database or only those that are concerned by the standard; to formulate this rule, we should first give a definition of one important term:

Imitating the relational data model, let’s say that a set of properties of an item forms its ***primary key****, if in the real world (or in some restricted environment where we operate) this item can be unambiguously identified by this set of properties.*

For example, a binary interface exported by a library written in C language is unambiguously defined inside the library by the name of appropriate source-level item (either function or global variable). That’s why name of the binary symbol of C library is equal to the name of appropriate source item (to be honest, with a small reservation for symbol versions). For C++ language, source-level name is not enough – one should add names of all namespaces and classes to which the item belongs (since functions with the same name can exist in different classes), as well as list of parameters (since every function can be overloaded, i.e. one class can provide several functions with the same name, but with different parameters). That’s why names of C++ binary symbols are constructed in a special way (by means of the process known as *mangling*) to reflect all these traits. In any case, if we consider the whole system, we should add name of the library to the primary key of every interface to unambiguously identify it; if we admit that the system can provide several libraries with the same name in different locations, then we should also add a library location, and so on.

By means of the primary key definition, we suggest to use the following rule to identify the properties that should be stored in the database:

Advice 1. *If it is detected that at least part of the primary key of some item should be stored in the database, then all properties of this item (including those that form the primary key) should be stored there. Otherwise, it is enough to store only those properties that depend on the standard.*

This rule is based on the fact that if we can get the primary key of some item without consulting the standard, then we can unambiguously detect the place of this item in the code of the tool to be generated. So we can simply put the item to the appropriate place in the tool template, leaving holes for secondary properties affected by the standard (if any). Since the location of item can be calculated, then it will cause no problems to create a generator that will detect the holes and populate them with data.

In other case, we can’t detect the actual place for the item in the code of the tool. Thus, we have to store all information about this item in some external storage and pick it up when necessary. Moreover, since we don’t know the standardized primary key of the item, we can’t exactly identify the item, and in general we should store information for all primary keys that are acceptable by the standard requirements. Fortunately, in many particular cases these possible primary keys are restricted to a very limited set. For example, as we saw above, in case of functions exported by libraries written in C language, a function name (maybe together with a library name) is enough – that is, there is no need to create several entries in the database for the same function, since all its attributes (parameters, return value) are unambiguously identified by the name.

One may notice that the database itself needs some primary key to store secondary properties of every item. This primary key can be exactly the same as the one of the item, or some artificial key that allows to calculate the ’real’ one (for example, a result of some hash function can be used to achieve a numerical artificial key to decrease the size of the database and to speed up the database queries). So the gain of not storing some properties in the database actually concerns only those properties that are not included in the primary key and don’t depend on the standard. In every particular case, this gain can be estimated; if it doesn’t prove to be significant it may make sense to store all properties of the item in the database.

Thus, if we are not sure if some item is obligatory for the tool and will appear there regardless of other conditions, then we should store all properties of this item in the database. But how generator will know if a given database entry should be picked up during generation? The simplest way is to assign a boolean flag to every database record:

Advice 2. *If all properties (including the primary key) of some kind of items are stored in the database, then additional boolean flag should be attached to every entry corresponding to this kind of items to indicate if a particular entry should be picked up by the generator when creating the tool.*

Let’s call this flag as *appearance* flag. With such a flag, generators should only select those entries for which the flag is set.

B. Handling Item Interdependencies

The most straightforward way to set the appearance flags is to set them manually. However, in some cases this can be a rather complicated task – the thing is that different items can depend on each other, and such interdependencies should be taken into account. For example, if we want to declare some function in a header file, we should also declare all types necessary for its declaration (that is, for parameters and return value) or to include other header files that will provide necessary declarations. The required types can have complex structure and require other types to be declared, and so on. Resolving such dependencies manually is tiresome; however, usually these dependencies can be clearly formulated, and it makes sense to automate the resolving process.

In order to do this, we suggest to create a *markup* tool, that will check and resolve dependencies between different items. In general, such a tool should be provided with a basis – a set of items that should be included in the generated code. This set should be formed manually and should contain all items that are included in the specification. On the basis of this information, the markup tool will decide which additional items should be included to the generated code to make it complete.

For example, let's suppose that the specification defines a set of C functions with signatures, and in the database we store function names, as well as type names, and mapping between function parameters and types, as well as mapping between return values and types. In order to generate fully qualified declaration, developers may only mark necessary functions as included, and the types that should be marked as included will be calculated automatically by the markup tool.

In this example we suppose that types are *subordinate* objects – if a type is required for some functions, then it is included without any discussion. However, another situation is possible, when the items in question are equal in rights – that is, it is not clear, if we should include a type or exclude all interfaces that use it. In case of such situations, the markup tool can only report the problem, but not solve it.

From the generator architecture point of view, there are at least two ways of implementing markup functionality:

- Implement a separate tool that will set flags directly in the database.
- Implement necessary functionality in generators; in this case all dependencies are resolved during generation, and database modifications are not required.

It's hard to say if one of these approaches is better than the other. The disadvantage of the second approach is that it makes the generation logic more complicated, and can slow down the generation process. On the other side, the first approach implies that the markup tool should be executed after any database change that can affect the generated code. For confidence, one can execute this tool every time the generation is launched, so the actual time of generation will also increase. The first approach is more preferable in case when the markup tool cannot resolve dependencies by itself and should interact with the user.

One more important question is how to store information about dependencies (that is, how the markup tool should know that one item depends on another). The straightforward way is to hardcode all the dependencies in the markup tool. However, there is a more elegant solution – if the used DBMS supports *foreign keys*, then it is enough to set such keys for the necessary tables, and the markup tool will be able to use them to detect dependencies. Considering again the example with C functions, one may store functions and their return values in the following way:

- Function names are stored in the *Function* table.
- Type names are stored in the *Type* table; let's suppose that this table has a primary key *Tid*.
- The *Function* table should also have *RetValue* field to store the type of its return value. This field should be a foreign key referencing appropriate *Type* record by its primary key *Tid*.

In general, we believe that this way is much more flexible than the hardcoded dependencies in the tools. So, one more rule:

Advice 3. *Interdependencies between different items should be implemented as foreign keys of appropriate tables in the database.*

C. Populating Database With Data

To design the database schema and to create generators is only a half of the problem. The database is useless until it is populated with data. We have already considered one aspect of the data management by suggesting a tool that will take care of item interdependencies, but this is just an auxiliary tool, a kind of consistency checker, though its usage can save a lot of time. However, the main problem is to collect the 'raw' data – that is, particular interfaces of different kinds with their properties and accompanying items. There can be no general method of solving such a problem, since in different cases the data can have different sources. For example, the whole set of standardized interfaces can be created from scratch by standard developers (e.g. as a result of some scientific research). In this case it is likely that there is no other way of populating database with data except typing the data manually. Some kinds of automation can be available in any case – for example, transforming textual lists of interfaces into SQL statements; however, these textual lists should be also achieved somehow. But here we'd like to point out one important situation when the significant part of necessary data can be obtained automatically. We are talking about the case when the standard is not created from scratch, but is based on some existing system, or generalizes interfaces of several implementations.

Advice 4. *In order to populate the database with data, it is useful to create a set of tools that will allow automated extraction of necessary information from existing systems.*

We suggest to extract as much data as possible from existing systems. We believe that there is no need to apply complicated filters to the collected data in order to select only those that have some chances to be useful for the standard. It is much more easier to set/unset the appearance flag for lots of entries than to collect additional data in case if one finds that we haven't collected enough. With the modern database management systems, it is unlikely that specification developers will reach some database limitations on the data size or performance – yes, the modern standards are large, and if we collect and store all the data that can be useful for the developers, we can achieve large amounts of data. For example, the LSB, one of the biggest interface standards in the world, contains about 40,000 interfaces. A usual Linux distribution consisting of one DVD disc can contain up to million interfaces, and theoretically, all these interfaces can be useful for the LSB. However, even these numbers are not a problem for the modern DBMS (in particular, for MySQL, used as the DBMS for the LSB Database which currently contains about 100 millions of records).

The task of data collection is a separate problem and can be even more complicated than generator development. As LSB developers' experience shows, it is much more simple to generate headers with declarations of functions and types on the basis of structured database information, then to parse and analyze existing headers in order to populate the database with this structured data. However, even this task doesn't seem to be very time-consuming when compared to the task of manual collection of data for 40,000 functions and similar quantity of types, constants and macros necessary for their declaration and usage.

Even more gain can be achieved if there are several tools that are generated (at least partially) using the database. Though these tools can require different generators to be created, it is likely that they use similar information from the database. If so, the same tool can be used to populate the database with data satisfying all generators.

D. Supporting Multiple Versions of the Standard

In the "Database Design" section, we've suggested to use an *appearance* flag to indicate that a particular item is included in the standard. To be sure, a single flag allows to store information corresponding to a single specification version. However, as we have noticed in the beginning, sometimes it is necessary to support several versions simultaneously. Forking a separate copy of the database for every version is a possible approach, but it introduces great data overhead (especially in case when different versions have significant intersection), complicates back porting of fixes for different issues and can significantly increase the maintenance cost of the whole system.

Another possible approach is to use the same database to store information about all specification versions. In this case the same tools can be used to generate data corresponding to a given version (moreover, it is possible to generate a single tool whose behavior can be adjusted by user to correlate with a particular specification version). Detailed description of possible approaches of improving the database to store such data can be found in [10]. Here we'll just summarize the main statements of that work:

Advice 5. *In order to store temporal data, one should replace the 'appearance' flag with the time interval that will indicate a set of versions where the item was included in the standard.*

This statement actually suggests to add a temporal dimension to the database. This can be done by using either temporal DBMS, or relational DBMS with additional fields indicating the interval bounds. With respect to specification versions, we may notice the following important features:

- The time is discrete; the possible values of interval bounds are standard versions, broadened with at least one specific value, that can be referenced as 'infinity'. If this value is used as a lower bound of the time interval, then this means that the entry has never been included in the standard. If it is used as an upper bound and the lower bound is not infinity, then the entry is included in all versions of the standard starting with those pointed by the lower bound of the time interval.
- When storing specification versions, one have to deal with only one kind of temporal data, called *valid time*, and don't need to store *transaction time*. That is, we should know in which version of the standard some item appeared, but not the time when the appropriate change was made in the database.

Thus, there is no need to track transaction time, and the set of possible values for the valid time is usually limited to a rather small set of values (fortunately, the standards doesn't introduce a new version every day). Moreover, a common life cycle of an item in the standard looks like *"appeared in version **A**, withdrawn in version **B**"*. In general, situations when some item is returned back after been withdrawn are quite rare. So for most cases the database will contain only one time interval indicated by two bounds – that is, if compared to a single appearance field, we just obtain one more field for every record. Thus, this approach is much more efficient from the data size point of view than a set of separate copies of the database corresponding to particular standard versions.

IV. Developing the LSB Development Environment

Let's now return to the LSB Development Environment and show how the ideas discussed above are used during its creation.

First, let's note that the main purpose of the Linux Standard Base specification is to pick out those interfaces provided by the operation system that are common to all major Linux distributions. The following kinds of interfaces are taken into account:

- Binary libraries (their runtime names).
- Binary interfaces (functions and global variables) provided by libraries.

- Commands (utilities and shell builtins).
- Modules of interpreted languages.
- Kinds of sections for the ELF files.
- RPM format tags.

That is, the LSB specifies the runtime environment where the application is running, and concerns the problem of distributing the applications in a compiled form. Applications that use only interfaces included in the LSB can be executed on any LSB compliant system without recompilation or environment adjustments.

LSB 4.0 contains specifications for 57 libraries with about 40,000 binary interfaces. These numbers might look large, but not when compared to any desktop Linux distribution – usual system on one DVD disk ships several thousands of libraries and up to million of interfaces. So it's not easy for application developers to orientate themselves in the Linux and LSB world. The developers who want to target LSB are supported with the LSB Navigator – web system that represents the LSB online, with lots of additional helpful information not included in the standard itself – and Linux Application Checker that can be used to check LSB compliance of executable files, shared objects and scripts that form the application.

However, these two tools are not free from the issues discussed in the beginning of this paper – consulting Navigator for every interface requires a lot of time; Application Checker can be integrated in the build process, but this will increase the build time, and in many cases the checker will just report the failure, but provide no suggestions on how to fix it. Application developers can meet some issues with obtaining LSB-compliant product when building their programs in the real systems, even if they don't directly use non-standard interfaces. For example, the following two problems arise quite often:

- LSB doesn't include arithmetic routines of the libgcc_s library. However, the gcc compiler forces usage of these routines if they are provided by the libgcc_s library that participates in the build process.
- Default behavior of the gcc compiler on some systems leads to the usage of ELF sections that are supported only by the last generation of distributions and thus not portable and not yet included to LSB. For example, Avinesh Kumar in his blog [11] explains why binaries compiled on RHEL 5 with the default compiler options will fail to run on RHEL 4.

Without a good knowledge of compiler operational principles, it's not easy to find out the roots of such issues.

To save application developers from these problems, it was decided to support them with the specific build environment, whose usage in the build process will guarantee that the obtained executables are compliant with LSB. At the moment two projects exist that provide such possibility:

- LSB Sample Implementation – the whole distribution built using the rPath technology that tries to limit provided interfaces to LSB ones.
- LSB Software Development Kit – a set of tools that can be installed in any real system and used as alternatives to the system build toolchain.

LSB Sample Implementation (LSB SI) is useful not only for building applications, but also for testing them, since it is guaranteed that implementations of its interfaces are compliant with LSB. However, since LSB SI is a distribution, it requires a separate machine to run; though it can be also used as a chroot environment, but even this variant is not very convenient for many developers. Another problem was already mentioned in the Section 2 – the development process of the SI is rather complicated and it is not guaranteed that it provides no forbidden interfaces at all.

On the opposite, LSB SDK doesn't suffer from these issues. To understand the reason, let's first consider the structure of this environment. It consists of the three major components:

- Stubs for libraries specified by LSB. These stubs export only those symbols that are included in the standard, but they don't provide their implementation.
- Header files that provide API (function declarations, types, constants, etc.) for the LSB libraries. It is guaranteed that usage of this API cannot lead to incompliant application.
- Compiler wrapper – a tool that should be called instead of system compiler. This tool calls the system compiler itself, forcing it to use libraries and header files provided by the LSB SDK. The environment variables and compiler options are automatically set to eliminate possibility of obtaining incompliant applications.

The advantage of the LSB development process, from our approach point of view, is the existence of *specification database* that stores different information about elements included in the specification. In particular, the database stores names of included libraries, as well as names and signatures of included interfaces. This database is used to generate LSB specification text and some primitive tests, and also serves as a knowledge base for LSB Navigator and Linux Application Checker. The data collection process that populates the database with information is separated from all other tasks and can be modified without affecting any other items of the LSB infrastructure. Thus, no wonder that the idea raised to use the database to automate the process of the LSB SDK development by introducing an automatic generation of some parts of the SDK, especially stub libraries and header files, that hardly depend on the entities included in the standard.

Stub library generation process has been implemented in a rather straightforward way. In order to obtain a desired stub, a source file in C language is generated with declarations of functions whose names are the names of binary symbols to be exported. Return types and parameters don't matter anything – we may use **void** as a return type for all functions and totally omit parameters. After this file is compiled, we'll obtain a shared library that exports exactly those symbols that are required by LSB.

This approach uses the fact that for libraries written in C, names of exported binary symbols (that are taken into account by the dynamic linker) are equal to the names of functions and global variables implemented in the source files; return types and parameters are not taken into account. For C++ and other languages that allow users to override functions, the situation is different – names of binary symbols are constructed during the *mangling* process on the basis of the function name itself, its parameters and name of the class or namespace which it belongs to. However, if we know binary symbol name of a C++ function, we may create a C source file with declaration of a function with this binary name and compile it. The result will be the same as if we create a C++ source file and place a proper function declaration there (i.e. with all necessary classes and parameters).

Thus, all that we should know to generate stub libraries are binary names of symbols that should be exported by them. The database fairly provides us with this information. Since the generated source files don't contain any actual code, then there are no interdependencies among functions that should be taken into account.

Unlike library stubs, header files should provide complete declarations of functions. All non-intrinsic types used in these declarations should also been declared. In addition, it is useful to declare such elements as constants and macros. Though are they are out of LSB scope, they can be useful for developers (but one should ensure that the macros don't invoke forbidden interfaces). Thus, though we still don't need function implementations, we have to declare complex types that sometimes have rather tricky interconnections, so generation of correct header files is a more complicated task then generation of library stubs. In case of C++, we also have to deal with templates that can affect runtime dependencies of applications. Every template should be analyzed to decide whether its usage can lead to calls to forbidden functions. This task was found to be rather complicated, and it is not completely solved at the moment, i.e. it is not guaranteed that C++ header files shipped with the LSB SDK cannot lead to usage of non-LSB functions. However, even if the latter happens and application obtains a dependency on a forbidden interface, this will be caught at the linking stage which will fail, since stub libraries don't provide the forbidden interfaces.

In the very beginning, the LSB database was populated manually; at that time, no one thought about SDK generation, and for the specification and test generation purposes the data was complete enough. As LSB evolved, the size of necessary data became too large to be handled manually, and additional tools were developed to automate data collection process (import* scripts, libtodb tool [12]). These tools were mainly based on the analysis of binary library files with debugging information, with slight header files analysis. However, though debugging information from binary files could give enough data to generate specification text and some tests, a lot of manual adjustments were required to make generation of header files possible. To improve data collection process, a new set of tools called LibToDB2 were developed at ISP RAS under the contract with the Linux Foundation [13]. The new tools analyze both binary files and headers, extracting data more accurately then their predecessors. The tools collect and upload to the database all information about analyzed headers and libraries. Then on the basis of lists of functions and global variables included in LSB, a separate tool discovers the types that should be present in header files to declare included interfaces. Surely, LSB workgroup can point out that some additional types are useful for developers and should be included in headers; these types should be marked as included manually. The same situation is with macros and constants, that are mainly selected manually, though there is a tool to check that included macros don't require non-LSB interfaces.

One more challenge was to make the LSB SDK to be a multiversion tool, that is, able to generate code complaint with any given LSB version. The LSB database contains data about all released versions of LSB (as well as about the one under development), so it can give us all necessary information about LSB history [10]. Thus, the task was to generate headers and stub libraries that could be used to target any LSB version. Depending on some environment conditions, the SDK should behave as if it represents a particular LSB version. The actual tool that user deals with is a compiler wrapper; to target a particular LSB version, user should only specify an option for it or to set appropriate environment variable, and the wrapper should perform all other necessary actions.

Since the set of symbols exported by a particular library cannot be affected by its environment, we can't generate a single stub file for every library to target several LSB versions. Instead, we create a separate file for every LSB version where the library appears, and let the compiler wrapper to choose the file to link against at runtime, on the basis of its options and environment variables. Unlike binary libraries, the contents exported by header files can be manipulated using different preprocessor directives. In the LSB SDK, every declaration is embraced with conditions on the LSB_VERSION constant, which can be set either directly by user or by the compiler wrapper. This results in smaller total size of files with respect to the case when a separate header is generated for every version, since we can create only one declaration of element to target several versions, and most elements are present in more than one version of LSB.

To estimate the gain achieved by usage of generators instead of manual creation of the whole SDK, one can compare the sizes of generated files with sizes of those part of the SDK that are written manually and with the size of generators themselves. Table I contains characteristics of the generated C-language part of the SDK (as was mentioned above, for C++ elements only stub libraries are generated at the moment, and generators of header files are under development).

Table II contains the same characteristics of the generators and data collection tools, which are written in Perl.

Table III contains the characteristics of the LibToDB2 set of tools (those part that collect C-language data), which is also important for the SDK generation. Note, however, that the data collected by these tools is used by many generators, not only by the SDK ones.

TABLE I
LSB SDK GENERATED CODE CHARACTERISTICS

	Stub libraries	Headers
Source code size, loc	87,800	68,500
Development effort estimate, person-years	22	17
Total estimated cost to develop, dollars	2,950,000	2,300,000

TABLE II
LSB SDK GENERATORS CODE CHARACTERISTICS

	Stub libraries	Headers
Source code size, loc	400	2,100
Development effort estimate, person-years	0,07 (1 month)	0,5 (6 months)
Total estimated cost to develop, dollars	10,000	60,000

TABLE III
LIBTODB2 CODE CHARACTERISTICS

Source code size, loc	4,400
Development effort estimate, person-years	1
Total estimated cost to develop, dollars	126,000

The data is generated using David A. Wheeler's 'SLOC-Count' [14] tool. Estimates are given by this tool using the basic COCOMO model. Surely, the estimates suppose that the code is created from scratch and don't take into account the nature of the code (such as absence of function implementations in stub libraries). One may also note that the SDK code can be created on the basis of appropriate upstream code by dropping those parts that are incompliant with LSB. However, such an analysis is rather difficult, and in any case it's not simple to support more than hundred of thousands of lines of code manually.

Schema of the database tables that are used during the LSB SDK generation is shown at Fig.1. Temporal data is stored in the tables that implement many-to-many relationships between entities and hardware architectures, since appearance of an entity on some architecture is independent from its presence on other platforms, as described in [10]. The only exception is header files – if a header is included on one platform, it is automatically treated as included on all others (even if it is empty there). So there is no direct relationship between headers and architectures, and temporal data is stored in the Header table itself.

Starting with LSB 4.0, the LSB SDK can be used to create executables and shared libraries compliant with any given LSB version, greater or equal to 3.0 (earlier versions are not supported by the database, since there was no great demand for them). At the moment, the SDK allows to generate only files fully compliant with LSB, with no exceptions (that is, one cannot use libraries or functions that are not included in LSB). In future it is planned to implement a *relax* mode for the

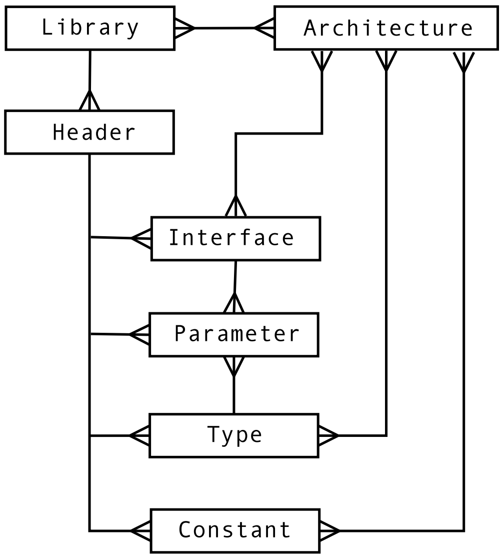

Fig. 1. Main Tables of the LSB Database Used During LSB SDK Generation

SDK that will allow applications to use certain symbols not included in LSB, but still known to be stable and present on most systems. Such possibility should increase the portability of applications (though formally the programs compiled in the relax mode will not be compliant with LSB). But even at the current state of art the SDK is very useful, and is used to build all programs involved in the LSB infrastructure (including test suites, LSB Application Battery and the SDK itself). Another good example of the SDK usage is the OpenPrinting project that suggest to write all printer drivers using the LSB DDK (Driver Development Kit) which is actually an enhanced version of the LSB SDK (supplemented by tools and libraries necessary for driver development but not included in LSB – such as Ghostscript or CUPS DDK) [15]. All current distribution-independent driver packages in the OpenPrinting database are based on LSB 3.2.

V. CONCLUSION

When creating a portable application, developers are interested in using only those interfaces that are present on the most of the target systems. One of the way of achieving this is to use interface standards that are followed by the target systems. However, direct consulting with the standard text is not very efficient, and consulting appropriate specialists can be expensive; in both cases, the development period can increase significantly. That's why it is important to support developers with toolchains whose usage will guarantee that the final product is compliant with the standard. However, integration of new tools can be also expensive and requires some time. So it is important to provide software developers with the tools similar to those that are already used in their development process, to make it cheap to replace existing tools with the new ones (or use the two toolchains in parallel). One of the possible ways of creating such tools is to modify the existing ones.

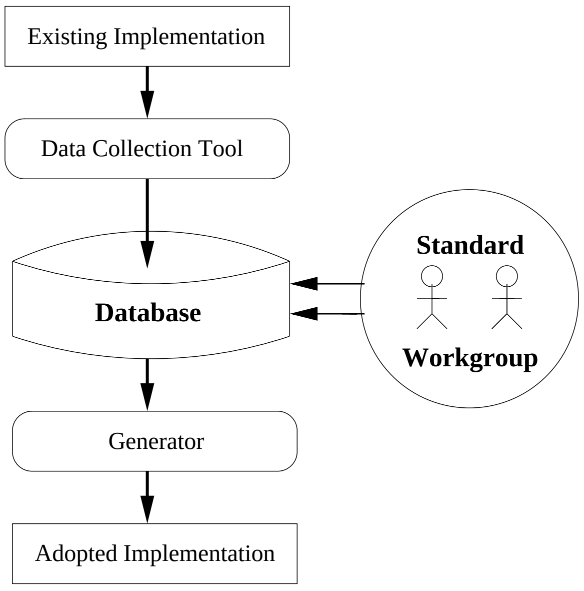

Fig. 2. Developing Adopted Tool Using Existing Implementation

This paper discusses an approach that allows to automate development and support of the tools based on the existing implementations. Fig.2 illustrates the suggested organization of the development process. The main idea is to collect data about existing implementation and to put it to the database in some intermediate format convenient for the specification developers. The data can be manipulated by developers in many ways; in particular, they can use special markers to separate those data that should appear in the tool from undesired or useless information. The automatic generators will use the adjusted data to create the tool itself; the tool achieved is actually a modification of existing implementation, adopted for the specification requirements.

Usage of the database allows specification developers to deal only with those properties of the tool that are concerned by the standard. All other aspects necessary for achieving a working program are handled by automatic generators that also propagate every change made in the database to all places where it should take effect. Thus, the specification developers can even have slight knowledge about the generated tool structure – they just should point out which interfaces are allowed, and which are not. Surely, complex interface standards require creation of complex developer-oriented tools, and the more complicated is a tool, the more complicated are appropriate generators and data collection tools. Nevertheless, development of complicated generators can be much more cheaper than development of the tools from scratch. And even more gain can be achieved if the same database is used to support several versions of tools corresponding to different versions of standards. Adding temporal dimension to the database and support of this dimension by the generators is not as expensive as support of several distinct tools or several distinct databases.

The approach suggested in this paper is used to develop the LSB SDK and proved to be useful and efficient, saving lots of efforts and resources. Since the SDK functions in a similar way as the system compiler toolchain does, its usage is transparent for developers and it is now adopted by many people who care about building software products portable among different Linux distributions in the form of binary files.

References

[1] Linux Standard Base. http://www.linuxfoundation.org/en/LSB
[2] Building Applications with the Linux Standard Base: Using the LSB Sample Implementation. http://www.linuxfoundation.org/en/Book/HowToSI
[3] SGI's Sample Implementation of the OpenGL API. http://oss.sgi.com/projects/ogl-sample/
[4] Building Applications with the Linux Standard Base: Using the LSB Development Environment. http://www.linuxfoundation.org/en/Book/HowToDevel
[5] All About the Linux Application Checker. http://ldn.linuxfoundation.org/lsb/all-about-linux-application-checker
[6] Becky S. Chu. appcert: A Static Application Checking Tool.Sun Developer Network, June 2001. http://developers.sun.com/solaris/articles/appcert.html
[7] Android Emulator. http://code.google.com/android/reference/emulator.html
[8] Linux From Scratch. http://www.linuxfromscratch.org/
[9] rPath. http://www.rpath.com
[10] Denis Silakov. Tracking Specification Requirements Evolution: Database Approach. Proceedings of the First Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE'2007), Volume 2, pp. 15-22. May 31 - June 1, 2007. - Moscow, Russia.
[11] Avinesh Kumar. Binary incompatibility between RHEL4 and RHEL5. http://avinesh.googlepages.com/binaryincompatibilitybetweenrhel4andrhel
[12] LSB DB Tools. http://ispras.linuxfoundation.org/index.php/LSB_DB_Tools
[13] LSB Library Import Tools. http://ispras.linuxfoundation.org/index.php/LSB_Library_Import_Tools
[14] SLOCCount Tool. http://www.dwheeler.com/sloccount/
[15] OpenPrinting: Writing And Packaging Printer Drivers. http://www.linuxfoundation.org/en/OpenPrinting/WritingAndPackagingPrinterDrivers