Platform-Dependent (Cost) Impacts on Portability, Software Reuse and Maintenance

'DASIA 2001'

- Data Systems in Aerospace -

Nice, France

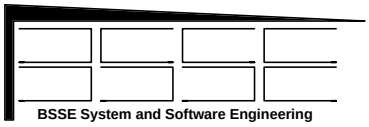
May 28 - June 1, 2001

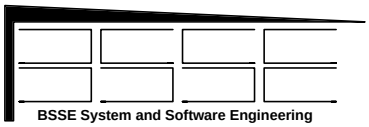
Rainer Gerlich

BSSE System and Software Engineering

Auf dem Ruhbuehl 181
D-88090 Immenstaad
Germany

Phone: +49/7545/91.12.58
Mobile: +49/171/80.20.659
Fax: +49/7545/91.12.40
e-mail: gerlich@t-online.de
www: http://home.t-online.de/home/gerlich/





Platform-Dependent (Cost) Impacts on Portability, Software Reuse and Maintenance

Rainer Gerlich

BSSE System and Software Engineering

Auf dem Ruhbuehl 181
D-88090 Immenstaad, Germany

Phone +49/7545/91.12.58
Mobile: +49/171/80.20.659
Fax +49/7545/91.12.40
e-mail: gerlich@t-online.de
URL: http://home.t-online.de/home/gerlich/

Abstract: Issues of portability, software reuse and maintenance are mostly considered as a matter of the software under development. This paper focuses on the impacts imposed by the software platforms on which the software is developed and executed, e.g. problems introduced by new tool versions or when moving to another tool or platform. Such problems arise because interfaces are changed, are not (fully) compatible or work-arounds for bugs had temporarily to be implemented. They are causing a lot of additional effort, costs and scheduling delays which cannot be estimated in advance. Also, it is nearly impossible to protect against such risks by development standards and guidelines at reasonable costs. As the use of Un*x [1] -based platforms based like Linux is steadily increasing, it is of high importance that decisions on the interface of such platforms are carefully done and potential impacts on users are considerd in advance. This paper wants to address the principal problems and to motivate for activities towards higher interface stability and conformity.

Keywords: Portability, reuse, maintenance, development costs, maintenance costs, C language ambiguities, Un*x shell weakness, deviations from standards, risk management

---

1. INTRODUCTION

It is believed that problems related to portability, software reuse and maintenance need to be solved by development standards and guidelines. However this is only true in part. Guidelines and standards can only cover known problems, but not problems actually arising e.g. when moving to a new version which includes changes which are not visible for the user. It is mostly impossible to identify all such problems in advance, e.g. by reading the documentation, even if carefully done. Usually, the description of the interfaces cannot cover everything what is important or may become important. Also, the reader may interpret ambigious text in his own way depending on what he is expecting or is considering as state-of-the-art. He even may not feel that such text is ambigious because he is biased on the interpretation by his own background.

E.g. the statements "platform XYZ provides standard BSD socket calls" and "you can use sockets that are source-code compatible with BSD 4.3 UNIX [2]" may lead to the (wrong) conclusion that everything is provided what is defined in the context of BSD sockets. Actually, this interpretation turned out to be wrong during later development because e.g. the feature of an asynchronous signal handler was not supported which was needed for efficient implementation of multiple input streams. From the tool vendors point of view it was covered because he provides an alternative solution which however is not fully compatible.Therefore an own solution had to be added which caused additional costs and scheduling delay.

On the other side, a software engineer cannot doubt everything what is described, and he usually does not have all the time to read the documentation in detail and to check if everything which is included e.g. in the POSIX [1] standard or in a Un*x environment, is actually provided, if he understood the documentation really correctly. But - according to Murphy's law - there is a good chance that a certain feature will be missing when it is urgently needed.

---

[1] Un*x is used as a common term for all Unix platforms

[2] UNIX is a trademark of AT&T

The problems mentioned above may be classified as "problems of understanding". There is a further category of problems related to change of tool interfaces, e.g. renaming of command options on shell/script level as it happend recently in case of Un*x (Solaris [2] and Linux [3]). In case of the "ps -ef" command all occurences had to be replaced by "ps ax". The cost and time impacts are the higher the more this feature is used e.g. for automated supervision of the generation and execution environment.

It is well known that such problems exist in practice but their cost impact is hardly traced. When such a problem occurs there is no time to record the additional costs and to assign them to the problem, because everybody needs to urgently concentrate on the problem's solution.

It is believed that such problems induce a number of risks, take a significant part of the budgets and cause scheduling delays. As the loss related to such problems is not recorded - as far as it is known - no initiatives exist to systematically collect data on such cost drivers in order to prevent them for the future.

This paper lists only some examples out of the long list of problems which have been observed. But they should be sufficient to become aware of the principal problem and to think about how the current situation can be improved.

Chapter 2 will describe a number of problems and chapter 3 will discuss recommendations and potential solutions. Finally, chapter 4 summarises the principal issues.

2. OBSERVED PROBLEMS

The problems related to platform-dependent impacts on portabilty, reuse and maintenance can be classified - so far - as problems

1. of correct understanding of the supported features,
2. due to semantic ambiguities
3. of interface changes.

A number of such problems will be described. However: the fact that such problems occured for the listed platforms neither does imply that such platforms are worse than other ones nor that other platforms not listed here do not cause such problems. The problems just have been recorded according to the more or less random usage of a subset of software tools.

2.1 The Problem of Supported and Unsupported Features

These problems are caused by informal interface descriptions, the interpretation of the text by the engineer according to his background and expectations, and the impossibility to document all the available and unsupported features by a few pages.

They typically occur when a system is ported to another platform, e.g. from Un*x to VxWorks [4], or when a platform is used the first time.

- Example 1: Compliance with Un*x BSD

  The example on BSD sockets - as mentioned in the introduction - was identified when porting existing software from Un*x to VxWorks. For matters of performance optimisation the application used message queues for local communication and sockets for remote communication. As it is not possible to wait on both queues without loosing performance by continuous polling or increasing the response times, the feature for asynchronous signals handlers was used in Un*x (Solaris, Linux) . But it was not supported by VxWorks as it turned out later on., although the overview of the VxWorks documentation gave the impression that it would be supported.

  The problem arose because the text in the documentation was misinterpreted. The statement "source-code compatible sockets are available" does not imply that all features are supported. It only means that such features which are supported may be source-code compatible. The concepts of pipes or non-blocking I/O, which are provided instead, were not helpful from a performance and functional point of view. Therefore an own solution had to be implemented which unified local and remote communication and provided the needed performance, which - however - was worse than the performance of the Un*x environment. Consequently, in this case the non-real-time Un*x platform lead to better results than the real-time platform.

- Example 2: Compliance with C pre-processor standards

  On the Mac OS 9 platform [5] the MPW environment [6] was used at the beginning. A number of pre-procesor directives of type "#if a==b" were successfully used as it was done for the GNU gcc compiler [7] before. When moving to the CodeWarrior environment [8] such expressions were rejected and it was identified that braces ("(...)") are needed like "#if (a==b)" for certain cases. Consequently, an update of the source code was needed. As far it is known the C standard does not request for such braces. And it is still unclear when brackets are needed and when not. The braces have just been added when an error message was printed.

- Example 3: Script execution

  Two principal problems were identified regarding csh (C shell) Un*x scripts when moving between Solaris and Linux platforms:

  ✧ A script which is syntactically correct may run on one platform, but not on the other one

    There are hidden side-effects which may prevent successful execution like:

    Un*x systems are using mass memory to temporarily store environment variables. If this information is stored in the system partition, this partition may be shared with the printer spooler or message logging. Then it may happen that the execution of a script fails because no sufficient storage is available and the result of operations on environment variables is empty.

    It happened that grep hang on one platform while it worked correctly on another for the following reason: due to 2 binary characters at the end of the file (the characters occured in a comment and hence were not relevant at all) caused grep to consider the file as binary file on one platform, while on the other platform it was processed as text file. When considered as binary file no output was produced and the next grep missed the input specification. This happened within a large tree of scripts at a customer's site and it took a number of iterations to fix the problem.

  ✧ A script which is syntactically NOT correct, may run on one platform, but not on another one

    Such problems were observed e.g. in case of mixing or missing parts of the constructs "if-then-endif" and "for-end" (csh). Missing "endif" or "end" or wrong sequence of "endif" and "end" may not be recognised by Un*x csh. Obviously, the interpretation of such syntactical terms is not unique and rigorous.

  Especially, the interpretation of the statements at run-time seems to be a problem in case of csh, because e.g. a missing "then" is only detected when the related "if"-statement is executed. Due to the missing capability of script compilation, such bugs may remain undetected, especially in cases which handle exceptions.

  Due to stepwise identification of the bugs, scripts had to be maintained a number of times causing a total effort which was higher than for the case when all bugs would have been detected by one run. The impact is the higher the more the organisation of the development environment is based on scripts.

  The conclusion is: compared with C scripts allow a quick implementation of a certain functionality based on utilities like grep, awk etc., but it is a real challenge to get them free of bugs. This compensates the short-hand savings at the end.

  According to the current experience it is planned to replace the scripts and the Un*x utilities by own programs in a mid- and long-term perspective in order to get rid of such weakness and related risks.

- Example 4: Actually supported OS services

  In case of Mac OS 9 pre-emptive multi-threading (called "multi-tasking" in the context of Mac OS 9) is supported - as the documentation says. However, when this feature should be used, it was recognised that only the functions related to multi-threading itself like "create task" or "create semaphore" can be used in this multi-threading environment plus a few more purely computational functions. However, nearly the complete set of OS services and application functions is excluded from use.

  When the decision towards use of Mac OS multi-tasking services was done no doubt came up because full support of multi-tasking is state-of-the-art. And at the end it was hard to believe that the support is really so limited.

  Therefore an own solution had to be implemented which provides full multi-threading for all available OS services and user functions.

2.2 Semantic Ambiguities

Usually, an engineer expects that the same term stands for the same behaviour, functionality or service. The problems described below are caused by violation of this experience. In case of reuse it is essential that the environmental conditions do not change. Of course, this requires full knowledge on the implied assumptions, and this is exactly the problem. Most assumptions are not clearly identified, even the originator may not know which assumptions he did or he implicitly relied on. This is a matter of informal descriptions. Such problems may happen e.g. in case of OS services or capabilities of a programming language as listed below.

Here are two examples:

- Example 5: Same term at user interface, but different OS procedures behind leading to surprising results

  Debian Linux 2.2 [9] suports a graphical login facility in addition to the textual login-procedure as supported in the past. Although the procedure is called "login" in both cases, the graphical login differs from the textual login significantly for at least one important detail. However, as observed for Debian Linux 2.2 the default configuration included in this distribution does not consider this difference.

  In case of textual login the user-login-file (like ".login" in case of csh) is executed and the environment is established as defined by the user. However, when logging in via the graphical interface the user's login-file is not executed and the user gets an incompletely defined environment, which in worst case may be the basic system environment only.

  Hence, execution paths and environment variables are missing which are needed for proper execution. Consequently, the system will fail to working. This happened for a presentation and caused a non-recoverable fault so that the demo had to be canceled.

  For preparation of the demo everything was checked out before. So there shouldn't have been any risk. However, during this check-out the login on the laptop occured over the network (in order to have a larger screen), while for the demo the graphical login procedure was applied. This really was the only difference, but it was sufficient that the demo failed because the problem could not be solved at short-hand, as a deeper analysis was needed.

- Example 7: Same name, but different, context-dependent interpretation and results

  Reused code may fail to work in case of the "sizeof" C-compiler directive. This may happen if "sizeof" is applied to an array. Consider the two iterations of reading a line from a file:

  Example 7.1:

```
char str[100];
...

fgets(str,sizeof(str),fd);
```

  Example 7.2:

```
int  readLine(char *str, FILE *fd)
{
    ...
    fgets(str,sizeof(str),fd);
    ...
}
```

  In case of example 7.1 up to 100 characters can be read from the file, while for example 2 only up to 4 characters can be read. Why?

  The answer is: In case of example 7.1 we have a bounded array and the compiler has the full knowledge on the array's size. Hence, "sizeof" returns 100. In case of example 7.2 "str" is a pointer to an array. The compiler only knows the size of the pointer, but does not have information on the real size of the array. So it evaluates to 4.

  What makes this difference critical, is the reuse of the code, in the example above the "fgets"-statement. Consider the case that example 7.1 is sufficiently tested and already well working. When similar code is needed or if for matter of modularisation the existing code is moved into a function, it won't work properly any more. In consequence, reused code has to be tested again, and code reuse might become as expensive as developing the code from scratch.

What makes the situation even worse, is the inherent risk of failure. As the code is reused nobody would expect that it will fail, this is a conclusion by analysis based on experience and wrong assumptions. Usually, it is difficult - although not impossible - to identify that the context has changed. If this change is not identifed there is a latent, permanent risk, and this makes the situation such dangerous. Moreover, most style guidelines do recommend the use of "sizeof" to obtain consistent code, but in this case the use of "sizeof" will yield a bug.

From this point of view code reuse has - unfortunately - some major disadvantages if tests are not repeated because the code is considered as already well tested. This reminds of the Ariane 5 accident in 1996 when code was reused without testing it again in the new environment.

2.3 Interface Changes

These problems are suddenly coming up when a new version of a software platform is put into operation. Hence, they are related to maintenance. The following examples represent a small subset out of a much larger list.

- Example 8: Un*x commands
  - ✧ Un*x ps (process list)

    For previous Un*x versions a complete list of existing processes could be obtained by "ps -ef". Now, this changed to "ps ax". Unfortunately, "ps -ef" (and "ps ef" as well) is still supported but addresses a different feature.

    A number of scripts had to be updated to adjust the ps-command.
  - ✧ Un*x ls (file list)

    The format of the command output was changed. A number of scripts evaluating automatically the name from a long file list (by filtering) had to be changed because the file name is now provided by column 8 while it was column 9 before.
  - ✧ Un*x tail (print tail of a file)

    Solaris supports the option "-lr" to invert the order of the lines, while Linux by its latest versions requires the command "tac" in addition to "tail". As Solaris (at least V2.5.1) does not provide the "tac" additional measures were needed to achieve platform compatibility.
- Example 9: Work-arounds and bug-fixing
  - ✧ gcc compiler

    For the gcc compiler the option for generation of debugger information changed from "-g" to "-gg" (V 2.7) and back to "-g" for the latest version 2.9.

    Although, only warnings are issued in case the wrong option is given, the warnings may confuse a user or may prevent that he can identify more important "real" warnings. Therefore, an update of the scripts was needed reflecting the version actually used. The reason for such changes of this option is unknown.
  - ✧ Un*x grep

    grep does not work correctly w.r.t. the "-w" option for Solaris 2.5 and 2.7. This was the reason why it was replaced by the equivalent GNU utility. The replacement as such is not the problem, but the needed recording of the change of the configuration. Actually, it was not done because it was considered a minor problem. However, when the scripts were distributed later on, the update by the GNU version was forgotten for the new target. In consequence, the scripts failed on the target environment and it took a lot of effort and time to identify the real problem. Hence, tracking of the platform configuration should be an issue in order to get rid of such problems.

    Moreover, the identification of such a problem may depend on which paths a user has included in which order. E.g. if he includes the GNU path before "/bin" he will never detect this problem because he always will implicitly use the GNU grep. So he will never become aware that configuration tracking is needed.
- Example 10: Consistent set of files (gcc h-files)

  When moving from gcc version 2.7.2 to 2.8 and 2.95 compilation errors occured for the new gcc versions while they did not for 2.7.2. It turned out that the set of h-files of the later versions implicitly include other h-files with the same file name in different directories.

This was identified for the set of h-files related to sockets (socketio.h, in.h) and shared memory (shm.in, sem.in, ipc.in). In principle, it is not possible to compile a simple "hello world" program free of errors which includes socketio.h and in.h. The reason is that the same file name is used in different paths for the later gcc versions while the file contents is NOT equivalent.

It is common practice to prevent multiple inclusion of the same h-file by guards like "#ifndef <guard name>". However, because the files are not equivalent, different names for the guards are used for the different branches. Hence, when including a file explicitly and another file which includes a non-equivalent file of same name implicitly a conflict arises due to multiple definitions.

The C source files had to be updated to master implicit conversion of the h-files.

- Example 11: Paths

  A number of problems are related to variation of path names of Un*x environments because system files or utilities are stored in different directories.

  ✧ system directories

  Utilities may be stored either in "/bin" or "/usr/bin". Especially, the paths may differ between Solaris and Linux and between the Linux distributions.

  If an engineer wants to be sure not to execute an alias he may refer to the utility by explicitly specifying the path to the desired utility. However, in this case he will fail if the utility is not availale by the given path. Then the path name needs to be adjusted.

  ✧ h-files

  It was observed that h-files (even if only one version exists) are stored in different directories. E.g. VxWorks provides "sys/times.h" and "sys/time.h" while Solaris and Linux provide "sys/times.h" and "time.h".

  Moreover, include-files may implicitly be included on one platform, but not on the other. If starting onthe platform which implicitly includes a h-file, compilation of the source code on the other platform will show errors due to the missing include. Hence, the source files need to be updated when porting the software. As it is unknown which other include-file implicitly includes the relevant file, it is impossible to provide software which is fully portable because such bugs are dormant.

  In case of Linux "net/if.h" and "netinet/in.h" must be included while inclusion is not needed in case of Solaris.

  For Solaris "stropts.h" must be included while the file is not explicitly required for Linux.

  For Linux "linux/if.h" must be included, but not for Solaris.

  In case of Linux and gcc V 2.95 "sys/socketio.h" and "linux/in.h" must be included in addition, while not required for gcc V 2.7.2.

  These are only a few examples out of a large set of conditional inclusion of h-files. It should not be surprising that an engineer loses the overview on what he actually does include. Finding the right files to be included is mostly a matter of "trials and errors/compiler warnings".

  Although platforms like Solaris and VxWorks already provide compatible contents of directories by use of alias h-files, incompatibilities still exist.

3. CONCLUSIONS AND RECOMMENDATIONS

For category 1 "understanding of supported and unsupported features" and category 2 "semantic ambiguities" the situation may be improved by introducing of a higher degree of formalisation. It seems that the observed ambiguities in interpretation can only be solved by a more formal approach. For category 3 there seems to be a simple solution - provided that the deciding engineers are willing to accept this solution: the compliance with interface standards and long-term stability of such interfaces.

However, in any case, the systematic collection of data on such weakness of software tools and platforms is needed to become aware of this problem. Only by having sufficient information and by providing feedback to suppliers the situation may change and improve.

3.1 A Matter of Formalisation

The problems related to "understanding of supported and unsupported features" and "semantic amgiuities" seem to be a consequence of the "informal" approaches which are behind. While the "world of informatics" is usually considered as a formal world, this is actually not true. For the widely used platforms, only the programming language and the compilers are based on formal specifications. For these case only, violation of rules can be detected, and the source code can be rejected.

Tests on conformance with standards may be already performed in case of C, POSIX and Ada [10]. However, above examples show that programming languages are not only the elements which are needed to implement and maintain software. Therefore the whole platform should be a matter of certification.

Unfortunately, formal specifications may not fully be applied even in case of programming languages, which in turn causes a number of problems like it was described for C. The syntax of the C language provides insufficient means for formalisation and to express unambiguously the desired actions. Consider example 7 on string handling in C.

Firstly, the actual length of a string can only be determinned by a scan of the string at run-time, an empirical, informal procedure. Secondly, the language does not allow to distinguish between "size of an array" and and "sizeof of a pointer to an array" regarding compiler directives. Thirdly, - as a consequence of 1 and 2 - the compiler cannot identify the wrong use of a "size of array" operation on a pointer.

A good point is, that there already exist programming languages like Ada which tackle this problem on a higher formal level. But this does not help the large community of C users at all.

While programming languages do syntax checking and can identify wrong use of the language prior to execution (at least in most cases), the situation is much worse for script languages (like used on Un*x platforms) which are only interpreted. Especially, this is true for the csh. But as csh and sh are not equivalent regarding their functionality a user may need both shells.

As scripts cannot be compiled, an error can only be detected when the statement is executed. Moreover, no standard procedure seems to exist by which the script interpreters of Un*x can be tested prior to shipping. At least the observations listed by chapter 2 lead to this conclusion. Obviously, more functionality is provided than can be tested. Therefore compilable scripts and script test suites should improve the situation.

The intention of scripts is to automate sequences on operating system level. If continuous maintenance is needed for such scripts, if it is difficult to verify them and if it remains unclear how realiable they are, their contribution to automation is limited, however.

Regarding the interface description, it would be extremely helpful if a tool vendor would clearly indicate which features are actually not supported or for which features alternative solutions exist. Of course, from a commercial point of view this might be a problem, but from the user's point of view it is a "must". However, if the users do not insist on that the situation must change, nothing will happen.

In addition, the project's risks can be further reduced by pre-testing such features which are considered essential. But usually such critical areas are not known in advance: a feature will get the status "essential" when it is recognised that it is not supported.

3.2 Stability of Interfaces

For the problems related to interface changes there should be an easy solution - in principle - consisting of the following steps:

- all platform providers need to agree on a (de-facto) standard for the complete platform
  - ✧ in case of Un*x and parts of Un*x like BSD sockets such standards need to cover paths to utilities, h-files and other relevant software, not only the interfaces of the basic functions
  - ✧ especially, if a platform provider supports a certain feature like BSD sockets he either has to fully comply with the standards or to clarly indicate what is missing
- all platform providers and other involved parties must ensure long-term stability of the interfaces and avoid overloading of terms.

Compliance with above two requirements becomes the more urgent, the more such platforms are used. From this point of view the Un*x environments, especially Solaris and Linux can contribute a lot towards easier and more efficient maintenance if they comply with such rules.

Un*x is usually considered a commonly agreed standard. But the examples above demonstrate that this is not true. What seems to be standardised are the system functions (based on POSIX), including the C library, and the commands on shell level (although a number of dialects exist on this level). As shown above there is still a wide area where standardisation does not exist, and for which each platform provider can and does come up with an own solution. Harmonisation of the current environments would help to save a lot of effort and time for maintenance and reuse.

It is doubted that official standardisation e.g. by ISO will help to improve the situation. Such standardisation imposes a lot of paper work and delays and may prevent meaningful evolution within a reasonable period. E.g. Ada does an update by a 10-years cycle, SDL [11] by a four-years cycle. And the standardisation committees only address the language, but not the complete language platform.

What is needed is an ad-hoc agreement by all involved parties onto de-facto standards and rules. This should also become an urgent issue of education.

To agree on and to comply with such standards - even if no formal act exists which requires it - should be a matter of (professional) education, which should address such aspects more deeply. From the user's point of view it is really not acceptable why options are changed forward and backward (as for the gcc debug option), why commands are changed and overloaded (as for Un*x ps command), why paths to standard files cannot be unique (as it happended for gcc h-files). Possibly, a closer contact between users and decision-making people at platform providers is needed.

Also, it seems that such changes are made under the assumption that e.g. commands are only typed in manually, and not processed by a machine. The current design and maintenance of user interfaces really seems to imply that an engineer manually operates them. However, the use of scripts should increase in future in order to achieve higher efficiency and this makes the problem more important. A script follows a given procedure and cannot consider changes of syntax and semantics. Vice versa, not improving the situation will impact future progress in efficiency of software portability, reuse and maintennace.

3.3 Risk Management

The problems described above may seriously impact the development costs and schedule or the success of a certain activity like a demo. It seems to be reasonable that by risk assessment the impact of such problems may be limited. However, in case of platform software there are a large number of features which may be a risk, that it is impossible to estimate the risk correctly.

Such a detailed assessment would probably require much more effort than the solution of problems which may come up. As it is unknown what was changed, each feature must be considered as risky as long as it is not proven by tests or by use that there is no risk associated with it. This is as bad as the case when the platform is considered as free of risks. Even if information about changes would be provided it is nearly impossible to conclude by analysis and without risk that there will be no impact on the use of the platform in the project.

Hence, it becomes impossible to master such risks. Even if an engineer is on CMM (Capability Maturity Model) level "managed" he may fail in accurately estimating the additional costs and delay. Consequently, the capability "to manage risks" does not help so much because a quantitative estimation is nearly impossible: the engineer just knows that there might be risks, but he cannot really react on that in a good manner.

Some of the risks only occur once, but that does not help so much. When such a problem arises the consequences may be very seriously like for the demo (example 5) which failed or examples 1 and 4 for which alternatives need to be found (and fortunately they could be found). As a platform will evolve the next version may bear other risks. And the same is true when moving to another platform.

According to above experience, regarding portability and maintenance it seems to be impossible to estimate a platform's risks when the platform is considered as a black-box and risk analysis focuses only on the interface description and its interpretation.

Regarding software reuse the currently available means are insufficient for identifying bugs which are related to undesired and unexpected context changes. This requires re-testing of reused software and decreases its cost saving potential. Or - if re-testing is not done - it makes software reuse risky.

Being aware of above exprience no final conclusion on satisfying risk management is possible other than: from the point of project management the risk is neglectable

- when porting and reuse of software are really finished
- when the system has been accepted, and
- when the system is not under maintenance.

4. FINAL REMARKS

A number of problems seriously impact porting, reuse and maintenance of software. Such problems are introducing risks and increase costs and development time. Regarding improvements of the software development process it is highly desirable that such impacts are reduced in future. Two principal sources have been identified so far: insufficient or ambiguous information on the relevant interfaces, and insufficient standards and changes which are not needed or not useful from the user's point of view.

It seems that the platform providers and deciding engineers need to be aware what impacts their decisions really will have. It should be a near-term goal to make such dependencies visible and to convince the involved parties on the needed improvements.

As far as such improvements are pending, it will be impossible to make porting, reuse and maintenance of software easier and less risky. Especially, regarding the lack of software engineers and in order to prevent wasting of man-power resources improvements are urgently needed.

Finally, wouldn't it be a good idea to collect information about such type of problems and to publish them? Only what is tracked, can be improved. Therefore I would like to encourage everybody to contribute to such a database so that the principle problems can be identified and be tackled.

REFERENCES

[1] POSIX Standard, IEEE standard P1003, "Portable Operating System Interface",
IEEE Standards Department, see http://www.ieee.org

[2] Solaris is a trademark of SunSoft Inc. 2550 Garcia Avenue, Mountain View, CA 94043, USA

[3] For information on Linux and download of Linux software see e.g.
http://www.linux.org or http://sunsite.unc.edu/pub/Linux/kernel/linux

[4] TORNADO / VxWorks, WindRiver Systems, Inc. 1010 Atlantic Avenue, Alameda, CA 94501-1153, USA

[5] Mac OS 9 is a trademark of Apple Computer, Inc., http://www.apple.com

[6] MPW "Macintosh Programmer's Workshop" is a freeware product of Apple Computer Inc.
for information see http://www.apple.com

[7] GNU-GCC is copyright protected by Free Software Foundation Inc.
information can be obtained via ftp from ftp://gcc.gnu.org/pub

[8] CodeWarrior, C-Compiler, Targeting Mac OS,
Metrowerks Corporation, 9801 Metric Blvd., Suite #100, Austin, TX, 78758 USA http://www.metrowerks.com

[9] Debian is a Linux distribution, for more information see http://www.debian.org

[10] Ada Programming Language, e-mail: adainfo@sw-eng.falls-church.va.us, URL: http://sw-eng.falls-chruch.va.us

[11] SDL, ITU, Recommendation Z.100, Specification and Description Language, SDL, 1993. Blue Book, Vol. X.1, and appendices A, B, C, D, F1, F2, F3, http://www.sdl-forum.org/

[12] GNAT Ada Compiler, http://www.gnat.com