Adaptation of Large-Scale Open Source Software – An Experience Report –

Markus Pizka*
Technische Universität München
Institut für Informatik - I4
Germany - 80290 Munich
pizka@in.tum.de

Abstract

Within a long-term distributed systems project we repeatedly stumbled across the well-known yet difficult question to either implement from scratch or comprehend and adapt existing software. Having tried both ways allows us to retrospectively compare the effectiveness of "from scratch" implementation versus software evolution. By using the code bases of GNU GCC and Linux for the adaptation approach we gained valuable experiences with the comprehension and adaptation of large but sparsely documented code bases. In most cases, the adaptation of existing software proved to be by far more effective than implementing from scratch. Surprisingly, the effort needed to comprehend the existing voluminous source codes repeatedly proved to be less than expected. In this paper we discuss our positive and negative experiences and the various factors influencing success and failure. Albeit collected in an academic setting, the observations described in this paper might well be transferable to the maintenance of large-scale commercial environments, too.

1 Retrospective Case Study

Experiences with maintenance and evolution of software systems can either be collected by performing a goal-driven empirical study, such as to demonstrate that technique A is superior to technique B. Or, experiences are acquired as a byproduct of non-maintenance motivated work on a software system. The goal-driven approach allows to focus on certain aspects, eliminate noise, and delivers comparable and eventually even repeatable results. In contrast to this, experiences gathered as a byproduct of some other work can not be pre-planned and the various factors influencing the observations can usually not be differentiated in a sound way. Besides these disadvantages, a by-product experience has the big advantage of being unbiased by goals and expectations set in advance or a certain qualification. It clearly reflects the difficulties, attempts, success, and failure encountered in a realistic situation.

In this paper we present such an unbiased maintenance experience collected in the course of a long-term distributed operating system (OS) project [10, 22, 21]. Our initial goal was to implement a new OS. Although no team member had specific maintenance skills, we soon faced the challenge to comprehend and evolve complex low-level yet sparsely documented code, in particular GNU GCC [26] and Linux [13]. By this we collected first-hand maintenance and evolution experiences of more than 10 people-years.

Now, this work allows us to investigate the following important maintenance questions by means of a retrospective study:

- Is sparsely documented code comprehensible?
- What's more promising: build from scratch, provide a wrapper, or evolve existing software?
- . . . and how can a practically oriented systems research group survive the burden of software implementation?

Outline

Section 2 sketches the difficulties of developing system level software and the different choices available to cope with these difficulties. Afterwards, we first present our initial experiences with implementation from scratch and the development of wrappers in section 4. We discuss the subjective expectations of the developers and their objective success or failure. We then relate these results with our later experiences with the invasive adaptation of open source software in section 5. In section 6 we summarize

*Part of this work was sponsored by the German Federal Ministry for Education and Research (BMBF) as part of the project ViSEK (Virtual Software Engineering Competence Center).

our maintenance experiences and conclude with a perspective for future work in section 6.

2 Motivation

Operating systems (OS) along with their corresponding infrastructure, such as the compiler, linker, and runtime system, represent particularly complex software systems for several reasons. The operating infrastructure must provide extensive functionality, serve several different purposes simultaneously, deliver satisfactory performance combined with a high degree of reliability, and even enforce security. All of this has to be implemented through a uniquely large number of levels of abstractions spanning from high-level programming languages [2, 9] down to subtle and highly machine-dependent details of the hardware-level [18] and with compatibility to existing applications.

Because of this, developing innovative OS concepts, such as in [4, 5] requires particularly strong efforts. For a research project in the OS field there are basically four different strategies to deal with the associated high costs:

1. no implementation: develop concepts only
2. from scratch: implement a complete new system
3. wrapping: implement on top of what is already there
4. adapt: reuse if possible but modify as needed

While strategy 1 is only of theoretical interest, strategy 2 is applied frequently [1]. It apparently provides maximum freedom and complete control over the final product. In reality, the immense costs of developing complex system level software from scratch are often underestimated which leads to either project failure [12] or changed objectives, i. e. restricted freedom. The majority of system level R&D projects tries to avoid these troubles by applying strategy 3. The new concepts are implemented on top of an already existing infrastructure without changing it. Middle-ware extensions such as CORBA [19] or PVM [11] are examples for this strategy. The advantages of this approach, such as low cost and a high degree of compatibility, come at a price. First, the capabilities of wrapper are constrained by the features of the already existing infrastructure. Second, if the services of the underlying infrastructure do not directly match the needs of the new concept, then they have to be implemented with the help of workarounds. This deteriorates the quality of the outcome in many aspects, such as performance and maintainability.

Although performance, security, compatibility and further aspects pose particularly difficult problems to OS developers, most of theses aspects play an important role in many other systems, such as large scale information systems, too. We therefore claim that our observations described below are not limited to the OS field but can be transfered to many other environments, too.

2.1 Software Adaptation

The disadvantages of the wrapping strategy can only be eliminated by invasive changes to the existing infrastructure leading to strategy 4, adaptation. Here, existing artifacts are reused as far as possible but consequently changed and extended if needed. Software adaptation[1] offers a promising combination of the flexibility of the from-scratch approach with the low costs of the wrapping strategy.

However, adaptation is rarely applied in practice and there are only few experiences with it. We argue that the reasons for this originate in a lack of software maintenance techniques and education. Software adaptation requires strong skills in program comprehension, reverse engineering, program transformation, and testing. Most of these fields and there application for Product Lines, Generative Programming [7], and others are just emerging and there is still a long way to go before these techniques can achieve maturity.

2.2 Setting for a Long-Term Adaptation Study

Regardless of the numerous open questions, we gained experiences with software adaptation, in the context of the distributed operating systems project MoDiS. In this project we defined a new parallel and distributed programming language INSEL [29] and designed a completely new resource management architecture with radical changes to the runtime system, linker, and the OS kernel [22].

In our attempts to implement these new concepts we experimented with the four implementation strategies, mentioned above. Our first compiler used C as an intermediate language and a conventional C compiler to produce executables. The resource management architecture was implemented as a wrapper on top of an existing UNIX. Because of the drawbacks of these prototypes, that is lousy performance, we later on tried to implement a new OS kernel from scratch. We further on implemented a significantly improved compiler by adapting GNU GCC [26] and finally continued our work on the kernel by adapting Linux according to our specific needs.

Over the period of 10 years, 10 researchers and approximately 30 computer science major students with different personal backgrounds and skills contributed to the various implementations. By this, we were able to observe numerous approaches, problems, solutions, and attitudes towards adaptation and their final success or failure.

[1] synonymously used for evolution

3 Related Work

Although the OS field is not the most active fields in computer science research there were and are still numerous commercial and research projects, such as QNX, Linux, Mach, and L4 [13, 1, 18] or even the on-going evolution of Microsoft Windows or Solaris. All of this projects must deal with the challenges described above and choose an implementation strategy. Unfortunately, the experiences acquired in these large scale projects concerning software maintenance are usually not collected in a systematical way and published because these projects focus in technical improvements.

A significant part of Brooks [14] widely respected work on software project management is based on his personal experiences with the development of IBM's OS/360 product family. This demonstrates the value of reflecting unbiased hands-on experiences besides controlled empirical studies. In this paper, we follow Brook's approach by reflecting our software maintenance experiences gained in a long-time project originally aimed at technical issues.

Aspects of software evolution independently of certain types are intensively studied by Lehman [15, 16] and other researchers in the software maintenance and evolution field. Without any doubt, these works mark important milestones on the way to understand the dynamics of long-lived software systems. In this paper, we are less interested in the principles of evolution but discuss pragmatic ways to deal with existing systems; i. e. the how to evolve. Rajlich for example, proposes a staged software life cycle [23], with a development, evolution, servicing, phase out, and close-down stage. He states that this sequence is uni-directional, that is no system returns from servicing back to evolution. The open source software used in our work seems to oppose this point of view.

Techniques to assess existing software and to decide on whether to build from scratch or to re-engineering are discussed in the software re-engineering field. The Software re-engineering Assesment Handbook (SRAH)[28] provides a structured process of choosing among different re-engineering or close-down strategies. Our OS work stated prior to our awareness of re-engineering and assessment techniques. Thus, our re-engineering decisions used to be ad-hoc without particular software maintenance, re-engineering, or program comprehension skills.

4 Early Experiences

As in many other research projects we also started the implementation of our new concepts by developing new services on top of UNIX. Now, from a distant and software maintenance perspective, it is surprising that such an important decision is usually made without a thorough analysis of alternatives and consequences. In fact, there are only few and hardly disseminated methods for the systematic analysis of different implementation strategies, although this decision has a major impact on both, budget and properties of the final product. Now, taking a look back, a SRAH assessment might have indicated technical limitations of the wrapping approach if the technical analysis was detailed enough. But usually, neither the evolving technical details are known a priori nor are assessment processes common to system level software developers.

4.1 Compiler Wrapper

Because INSEL is an object-based language the idea to translate INSEL into C++ and using a C++ compiler seemed plausible. Thus, the first compiler for INSEL translated INSEL into C++ in a straight forward manner. It soon became evident that this approach provided insufficient control especially concerning details of stack management and register usage.

To circumvent these deficiencies, two years later, a new compiler was developed that translated INSEL into sophisticated low level C code with excessive use of address arithmetics [29]. While this new compiler eliminated most of the shortcomings of the C++ variant and C seemed to be highly flexible at first sight, new restrictions became apparent. For example, C does not support nesting of functions. Therefore, nested INSEL functions somehow had to be mapped on C functions by putting the local variables of the enclosing function into a `struct` and passing a pointer to the nested function. Other compilers like the *p2c* Pascal compiler use similar techniques. This minor workaround has already major impact on the performance. It entails a performance degradation of up to 30%, already [20]!

Wrappers often cause such compromises each of which entailing performance degradation or – to put it more general – loss of quality. The sum of this minor losses leads to serious decrease in quality. In our case a compiler generating target code at least two times slower than it should be.

4.2 Runtime Wrapper

Along with the two compilers, two runtime systems were developed on top of UNIX in C++ and C. As expected, this approach quickly lead to executable prototypes allowing experiments with the new concepts. These experiments allowed us to show that our resource management architecture is able to deliver perfect linear speed-ups with increasing numbers of processors.

Unfortunately, it could also be shown, for example that our parallel system based on C++ performed 700% worse

than sequential C code! Similar observations were made in other projects, such as in Orca [17] and CORBA [24].

The reasons for this are analog to the troubles of the compiler. The wrapping strategy entails severe quality degradations if the chosen basis does not perfectly suit the concepts implemented on top of it because workarounds are expensive both during development and at runtime. It is important to note that minor discrepancies may cause major disadvantages. We assume, that other quality attributes, such as security, reliability, and maintainability are affected in a similar way. Indeed, functionality seems to be the only aspect [6] that is unaffected by the wrapping strategy.

4.3 From Scratch Kernel

Based on the experiences with the wrapped runtime systems one of our groups started to develop a new, more suitable OS micro-kernel [8]. This group considered the possibility to implement the new kernel by modifying an already existing one. But, the group shared the common belief, that the source codes of existing open source kernels were insufficiently documented and comprehending the low level code by reading it would be just as expensive as writing a new kernel. They furthermore argued, that some of their concepts could hardly be implemented by changing the existing system, at all. Thus, after assessing different implementation and re-engineering strategies they concluded that the kernel had to be implemented anew from scratch.

The kernel project soon fell behind its schedule to deliver a working prototype within 2 people years. Design and implementation took multiples of the estimated times because technical details repeatedly proved to be a lot more difficult then expected. After 6 people years, the kernel was still far from being usable. The project was aborted. During this work a significant amount of work was devoted into "reinventing the wheel".

The reasons for the failure seem twofold. First, high-level concepts often entail conceptually simple but technically demanding details. It is easy to understand preemptive scheduling [25] at the conceptual level but it is far more complicated to understand the code of an optimized scheduler and it is even more challenging to write a new. From our own experience and further observations we believe that the complexity of the detail is often either underestimated or falsely ignored. Second, a huge amount of time was spent on implementing and testing concepts anew although they have already been solved in existing systems.

Taking common software engineering experiences into concern, such as software testing takes approximately 50% of development time, it should become clear that implementing from scratch is time consuming and error-prone even if the concept to be implemented is not new. Therefore re-implementations are to be avoided if there are no serious reasons to it. In the case reported here, this rule of thumb was not respected.

4.4 Summary

It should be emphasized, that the troubles described in this section were not caused by particularly weak skills of the project team. In fact, numerous commercial and academic system-level software projects run into similar difficulties for the same reasons. The lack of suitable maintenance techniques and education leads to prejudices against foreign code and the inability to evolve existing systems. This in turn results in either expensive "re-inventions of the wheel" or hard-to-maintain workarounds with weak quality.

5 Adaptations

Our own experiences with from scratch development and wrapping along with observations of numerous similar projects show that the impact of the costs of implementation on progress of a field[2] are stronger than expected.

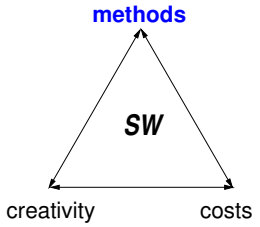

Figure 1. method-creativity-cost triangle

As illustrated in figure 1 we experienced and observed strong dependencies between methodology, creativity and costs. Insufficient methods not only increase implementation costs but the increased costs also restrict creativity because creative ideas are often abandoned quickly if the costs seem to be high. But without creativity there are no new methods.

With this interdependency in mind it is clear that first, the costs must be reduced to allow significantly advancements of the field, later. We therefore aimed at reducing the implementation costs without sacrifying the conceptual level by using and changing high-quality open open source software according to our needs. We chose the compiler GNU GCC as the starting point for the development of a new INSEL compiler and Linux [13] as the basis for our OS kernel. The results achieved with this maintenance approach exceeded our expectations.

[2] here, system level software

5.1 Compiler

Without doubt, GCC is one of the largest and most complicated open source software packages, available today. Despite the GCC mailing lists, some online archives and a rather general introduction [26], there is no comprehensive documentation of its implementation besides in-line code. The constantly growing core of the compiler exceeds 500.000 loc[3] and supports several languages of the C family on more than 200 platforms. At the same time, GCC is a high quality compiler. Apple's Max OS X (10.2) is compiled entirely using GCC 3.1 and companies such as SCO offer GCC as their standard compiler.

Because of GCC's enormous complexity, implementing our own new imperative language INSEL by changing GCC was considered very ambitious. Indeed, some parts of GCC represent the maintainers nightmare at first sight. Besides the absence of documentation, there are single functions, such as `expand_call`, with 1300 loc. Furthermore, different parts of the compiler are written in different languages, e. g. a bison parser specification, C interspersed with 100s of different macros and in-line assembler, and proprietary machine and hardware description languages. The C code makes excessive use of sophisticated C short cuts, such as pre and post increment/decrement, and the ternary operator.

5.1.1 First Steps

Because of this, we started the project to implemented INSEL as a new language of GCC cautiously. A single undergraduate student with at most average programming skills received the unappreciative task to read the GCC sources and find a way to implement INSEL on the basis of GCC. To our surprise, this work advanced much faster then expected. Within his 3 months thesis [27] the student had analyzed the architecture of GCC, understood its concepts on different levels and implemented a first prototype of the new GNU INSEL compiler GIC [20] capable of translating the function shown in figure 2 from source to any supported platform with arbitrary optimization switches.

```
FUNCTION foo(x,y:IN integer) RETURN R_T
  lokal: integer;
  R : R_T;
BEGIN
  lokal := x+y+a;
  R.X := Lokal;
  R.Y := Lokal+1;
  RETURN R;
END foo;
```

Figure 2. Function foo

[3] lines of code

Besides providing an executable compiler, we learned several software maintenance lessons within this thesis. In contrast to the wrapping and the from-scratch approaches, everyone in team overestimated the time needed to accomplish the tasks to read the existing code and to extend it. Code reading, understanding and modification proceeded much faster than expected. Opposed to common belief, foreign code proved to be only difficult to understand in the beginning. Once the code reader has gotten over initial frustration, started to accept the different style of the code, and found certain entry points code reading and understanding is very quick.

5.1.2 Subsequent Work

After the success of this initial work, several students continued development of GIC over 2 years. During this period we made two major observations:

1. Progress slowed down with increasing size.

2. The productivity varied enormously between different developers. The ideal skills are unclear.

Though observation 1 is not surprising it seems to set an upper bound for the successful extension of complex software in a distributed and open source manner. In fact, the core of GCC itself is maintained by a relatively small number of developers. Significant progress tends to occur only outside the core where dependencies to other parts of the system are limited. For this, it is essential that the code based is thoroughly modularized so that individual developers are able to operate with limited knowledge. Indeed, it can be observed that open source products are frequently restructured as needed to allow future growth [3].

The difference between the productivity of different developers was extremely high, although most participating students joined the project at a similar stage of their studies. While some students implemented complex language concepts, such as explicit task parallelism, within a few days, others felt unable to make any significant contribution after months of trying.

As of today, we can only speculate about the reasons for this. First, adaptation requires a broad understanding of different concepts. In our case details of C, address arithmetics, macro usage, assembler, compiler construction, etc. Hardly anything can be done if only one of these skills is missing. Second, the willingness and ability to read and to fully understand foreign code varies significantly. While some developers more or less refuse to read code not written in their style, others are willing to invest a large amount of time to understand the intentions and reasons of their predecessors.

5.1.3 Remark

The result of the GCC adaptation project was the fully operational and highly optimizing INSEL compiler GIC [20]. In contrast to the preceding INSEL compilers, GIC did not introduce any overhead. With GIC, INSEL achieves similar performance like optimized C programs.

5.2 Kernel

We later on continued our adaptation work by replacing the wrapped runtime system with a more suitable kernel by changing Linux according to our needs for an OS kernel targeted towards distributed and parallel computing.

Most of the GCC properties, such as large, complex, incompletely documented, multiple languages, etc. apply for Linux, too. Nevertheless, we also adapted Linux according to our needs in a series of theses. The kernel was extended with additional services, the management of task stacks along with the task data structure was changed, and the usage of registers was modified.

From the software maintenance perspective the kernel adaptation project reproduced the experiences gained in the compiler adaptation project although different students were involved in this project. Adaptation has been by far more successful than any other implementation strategy although the Linux kernel is highly complex, its code is far from being trivial to read, and adequate maintenance tools are still missing. Again, the individual skills of the different developers played an important role for the success of the adaptation work.

6 Conclusion

In the context described in this paper, the adaptation of existing software has demonstrated its superiority compared to development from scratch or wrappings. Our experience clearly shows that it is a myth that the understanding of different, foreign, critical, highly complex, and undocumented code for extension purposes was similarly difficult than writing a new system. Even the complicated low-level C source code of GCC and Linux could be analyzed and comprehended by students within weeks after which they were able to contribute changes and extensions.

The decision of whether to develop from scratch, to wrap or to change existing similar software has a major impact on the outcome. Provided that there are enough resources, it can be expected that from scratch development is able to deliver products with optimal quality. But in reality, resources are scarce and the time needed just for solving problems that are already solved elsewhere often exceeds the available resources. The effort needed for development from scratch tends to be underestimated because the upcoming complexity is not yet visible.

Wrapping provides a poor way out of this dilemma because mismatches between the actual goals and the provisions of the underlying basis necessitate workarounds. These workarounds have negative influences on the quality of the final product in many aspects.

The invasive adaptation of existing software, that is its consequent modification and extension, seems to be the only realistic solution in the long run. Obviously, this is not limited to the operating system context described in this paper but can be transfered to large scale commercial software systems as well. Interestingly, adaptation repeatedly proved to be less time-consuming than expected because the first estimation tends to be pessimistic due to scepticism concerning foreign coding style and the size of the unknown object.

As described in this paper the maintenance skills of the individual developers are of predominant importance for the success or failure of adaptation work. We believe, that the success described in this paper could be leveraged with advanced software maintenance techniques, tools and education in particular concerning comprehension and transformation of existing software.

Here, we presented our maintenance experience on a very coarse and descriptive level. Our next step will be to analyze the difficulties and phenomenons encountered during this long-term adaptation project in detail and to compare the resulting requirements with existing maintenance techniques in a systematic way. We also want to perform a retrospective SRAH assessment in the near future to analyze the possible impact of a structured assessment on our project.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. Technical report, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, August 1986.

[2] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[3] A. Bauer and M. Pizka. The contribution of free software to software evolution. In *Proc. of the Int. Workshop on Principles of Software Evolution (IWPSE)*, Helsinki, Finland, Sept. 2003.

[4] V. Cahill, R. Balter, D. Harper, N. Harris, X. R. de Pina, and P. Sousa. The Comandos distributed application platform. *The Computer Journal*, 37(6):478–486, 1994.

[5] R. H. Campbell and N. Islam. Choices: A Parallel Object-Oriented Operating System. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programmi ng*. MIT Press, 1993.

[6] X. Corporation. Aspect-oriented programming home page. http://www.parc.xerox.com/aop/, 2000.

[7] K. Czarnecki and U. W. Eisenecker. *Generative Programming*. Addison Wesley, 2000.

[8] C. Czech. Dycos - a customizable kernel architecture supporting distributed operating environments. In *Proc. of IEEE 3rd Int'l Conf. on Algorithms & Architectures for Parallel Processing (ICA3PP'97)*, Dec. 1997.

[9] C. Eckert and M. Pizka. Improving resource management in distributed systems using language-level structuring concepts. *The Journal of Supercomputing*, 13(1):33–55, Jan. 1999.

[10] C. Eckert and H.-M. Windisch. A new approach to match operating systems to application needs. In *IASTED – ISMM International Conference on Parallel and Distributed Computing and Systems*, pages 499 – 503, Washington, USA, Oktober 1995.

[11] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT press, 1994.

[12] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the USENIX Summer 1993 Technical Conference*, pages 147–160, Berkeley, CA, USA, June 1993. USENIX Association.

[13] G. Hankins. The linux documentation project. World Wide Web, 1999. http://www.go.dlr.de/linux/LDP/.

[14] F. P. B. jr. *The Mythical Man-Month*. Addison Wesley, 1995.

[15] M. Lehman. The programming process. Technical Report RC2722, IBM Research Centre, Yorktown Heights, NY, Sept. 1969.

[16] M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 2001.

[17] W. G. Levelt, M. F. Kaashoek, H. E. Bal, and A. S. Tanenbaum. A comparison of two paradigms for distributed shared memory. *Software– Practice and Experience*, 22(11):985–1010, Nov. 1992.

[18] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, Colorado, Dec. 1995.

[19] OMG. Corba services:common object services specification, security services. Technical report, OMG and X/Open Co Ltd., July 1997.

[20] M. Pizka. Design and implementation of the GNU INSEL-compiler gic. Technical Report TUM-I9713, Technische Universität München, Dept. of CS, 1997.

[21] M. Pizka. Distributed virtual address space management in the MoDiS-OS. Technical Report TUM-I9817, Technische Universität München, 1998.

[22] M. Pizka and C. Eckert. A language-based approach to construct structured and efficient object-based distributed systems. In *Proc. of the 30th Hawaii Int. Conf. on System Sciences*, volume 1, pages 130–139, Maui, Hawai, Jan. 1997. IEEE CS Press.

[23] V. T. Rajlich and K. H. Bennett. A staged model for the software life cycle. *IEEE Computer*, pages 2–8, July 2000.

[24] D. C. Schmidt and S. Vinoski. Object interconnections. *SIGS C++ Report magazine*, May 1995.

[25] A. Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts*. Addison Wesley, 3. edition, 1991.

[26] R. M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Nov. 1995.

[27] C. Strobl. Integration der sprache INSEL in den GCC. Fortgeschrittenenpraktikum, July 1996.

[28] STSC. Software Reengineering Assessment Handbook v3.0. Technical report, STSC, U.S. Department of Defense, Mar. 1997.

[29] H.-M. Windisch. The distributed programming language INSEL – concepts and implementation. In *First International Workshop on High-Level Programming Models and Supportive Environments*, pages 17 – 24, Apr. 1996.