To appear in IEEE Trans. on Software Engineering 1998 (Ref.no. S96134)

A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking

Lutz Prechelt (prechelt@ira.uka.de)
Walter F. Tichy (tichy@ira.uka.de)
Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe, Germany
Phone: +49/721/608-4068, Fax: -7343

March 23, 1998

Abstract

Type checking is considered an important mechanism for detecting programming errors, especially interface errors. This report describes an experiment to assess the defect-detection capabilities of static, inter-module type checking.

The experiment uses ANSI C and Kernighan&Ritchie (K&R) C. The relevant difference is that the ANSI C compiler checks module interfaces (i.e., the parameter lists of calls to external functions), whereas K&R C does not. The experiment employs a counterbalanced design in which each of the 40 subjects, most of them CS Ph.D. students, writes two non-trivial programs that interface with a complex library (Motif). Each subject writes one program in ANSI C and one in K&R C. The input to each compiler run is saved and manually analyzed for defects.

Results indicate that delivered ANSI C programs contain significantly fewer interface defects than delivered K&R C programs. Furthermore, after subjects have gained some familiarity with the interface they are using, ANSI C programmers remove defects faster and are more productive (measured in both delivery time and functionality implemented).

Keywords: type checking, defects, quality, productivity, controlled experiment.

1 Introduction

The notion of data type is an important concept in programming languages. A data type is an interpretation applied to a datum, which otherwise would just be a sequence of bits. The early FORTRAN compilers already used type information to generate efficient code for expressions. For instance, the code produced for the operator "+" depends on the types of its operands. User-defined data types such as records and classes in later programming languages emphasize another aspect: Data types are a tool for modeling the data space of a problem domain. Thus, types can simplify programming and program understanding. A further benefit is type checking: A compiler or interpreter can determine whether a data item of a certain type is permissible in a given context, such as an expression or statement. If it is not, the compiler has detected a defect in the program. It is the defect-detection capability of type checking that is of interest in this paper.

There is some debate over whether dynamic type checking is preferable to static type checking, how strict the type checking should be, and whether explicitly declared types are more helpful than implicit ones. However, it seems that overall the benefits of type checking are virtually undisputed. In fact, modern programming languages have evolved elaborate type systems and checking rules. In some languages, such as C, the type-checking rules were even strengthened in later versions. Furthermore, type theory is an active area of research [3].

However, it seems that the benefits of type checking are largely taken for granted or are based on personal anecdotes. For instance, Wirth states [21] that the type-checking facilities of Oberon had been most helpful in evolving the Oberon system. Many programmers can recall instances when type checking did or could have helped them. However, we could find only a single report on a controlled, repeatable experiment testing the benefits of typing [9].

The cost-benefit ratio of type checking is far from clear, because type checking is not free: It requires effort on behalf of the programmer in providing type information. Furthermore, there are good arguments why relying on compiler type checking may be counterproductive when doing inspections [12, pp. 263-268].

We conclude that the actual costs and benefits of type checking are largely unknown. This situation seems to be at odds with the importance assigned to the concept: Languages with type checking are widely used and the vast majority of practicing programmers are affected by the technique in their day-to-day work. The purpose of this paper is to provide initial, "hard" evidence about the effects of type checking. We describe a repeatable and controlled experiment that confirms some positive effects: First, when applied to interfaces, type checking reduced the number of defects remaining in delivered programs. Second, when programmers use a familiar interface, type checking helped them remove defects more quickly and increased their productivity.

Knowledge about the effects of type checking can be useful in at least three ways: First, we still lack a useful scientific model of the programming process. Understanding the types, frequencies, and circumstances of programmer errors is an important ingredient of such a model. Second, a better understanding of defect-detection capabilities of type checking may allow us to improve and fine-tune them. Finally, there are still many environments where type checking is missing or incomplete, and confirmed positive effects of type checking may help close these gaps.

In this experiment we analyze the effects of type checking when programming against an interface. Subjects were given programming tasks that involve a complex interface (the Motif library). One group of subjects worked with the type checker, the other without. The dependent variables were as follows:

Completion Time: The time taken from receiving the task to delivering the program.
Functional Units: The number of complete and correct functional units in a program. Each functional unit interfaces to the library and corresponds to one statement in the "gold program" (the model solution).
Interface Use Productivity: measured in Functional Units per hour and by Completion Time.
Number of Interface Defects: The number of program defects in applying the library interface. Such a defect is either an argument missing, too many, of wrong type, or at incorrect position; or it is the use of an inappropriate function.
Interface Defect Lifetime: The total time a particular interface defect is present in the solution during development. Note that this time may be the sum of one or more time intervals, since a defect may first be eliminated and later reintroduced.

We conjecture that type checking makes type defect removal quicker and more reliable, thus also speeding up overall program development. More concretely, we attempt to find support for, or arguments against, the following three hypotheses.

- **Hypothesis 1:** Type checking increases Interface Use Productivity.
- **Hypothesis 2:** Type checking reduces the number of Interface Defects in delivered programs.
- **Hypothesis 3:** Type checking reduces Interface Defect Lifetimes.

2 Related work

We are aware of only two closely related studies. One is the Snickering Type Checking Experiment[1] with the Mesa language. In this work, compiler-generated error messages involving types were diverted to a secret file. A programmer working with this compiler on two different programs was shown the error messages after he had finished the programs and was asked to estimate how much time he would have saved had he seen the messages right away. Interestingly, the programmer had independently removed all the defects detected by the type checker. He claimed that on one program, which was entirely his own work, type checking would not have helped appreciably. On another program which involved interfacing to a complicated library, he estimated that type checking would have saved half of total development time. It is obvious that this type of study has many flaws. But to our knowledge it was never repeated in a more controlled setting.

A different approach was taken by the second experiment, performed by Gannon [9]. This experiment compares frequencies of errors in programs written in a statically typed and a "type-less" language. Each subject writes the same program twice, once in each language, but a different order of languages is used for each half of the experiment group. The experiment finds that the typed group has fewer distinct errors, fewer error re-occurrences, fewer compilation runs, and fewer errors remaining in the program (0.21 vs. 0.64 on average). The problem with the experiment is that it was significantly harder to program with the type-less language. The task to be programmed involved strings and the typed language provided this data type, while the type-less language did not. Gannon reports that most of the difficulties encountered by the subjects were actually due to the bit-twiddling required by lack of typing and that "relatively few errors resulted from

[1] J.H. Morris, Xerox PARC, unpublished, 1978

uses of data of the wrong type" ([9], p.591). Hence the experiment does not tell us how useful type checking is.

There is some research on error and defect classification, which has some bearing on our experiment. Several publications describe and analyze the typical defects in programs written by novices, e.g. [6, 18]. The results are not necessarily relevant for advanced programmers. Furthermore, type errors do not play an important role in these studies.

Defect classification has also been performed in larger scale software development settings, e.g. [1, 10]. Type checking was not an explicit concern in these studies, but in some cases related information can be derived. For instance, Basili and Perricone [1] report that 39 percent of all defects in a 90.000 line FORTRAN project were interface defects. We conjecture that some fraction of these could have been found by type checking.

The defect-detection capabilities of testing methods [2, 8, 22] have received some attention; the corresponding psychological problems were also investigated [20]. There is also a considerable literature about debugging, e.g. [7, 13, 16, 17], and its psychology, e.g. [17, 19]. However, the defects found by testing or debugging are those that already passed the type checks. So the results from these studies would be applicable here only if they focused on defects detectable by type checking — which they do not.

Several studies have compared the productivity effects of different programming languages, but they either used programmers with little experience and very small programming tasks, e.g. [6], or somewhat larger tasks and experienced programmers, but lacked proper experimental control, e.g. [11]. In addition, all such studies have the inherent problem that they confound too many factors to draw conclusions regarding type checking, even if some of the languages provide type checking and others do not.

It appears that the cost and benefits of interface type checking have not yet been studied systematically.

3 Design of the Experiment

The idea behind the experiment is the following: Let experienced programmers solve short, modestly complex programming problems involving a complex library. To control for the type-checking/no-type-checking variable, let every subject solve one problem with K&R C, and another with ANSI C. Save the inputs to all compiler runs for later defect analysis.

A number of observations regarding the realism of the setup are in order. A short, modestly complex task means that most difficulties observed will stem from using the library, not from solving the task itself. Thus, most errors will occur when interfacing to the library, where the effects of type checking are thought to be most pronounced. Furthermore, using a complex library is similar to the development of a module within a larger project where many imported interfaces must be handled. To ensure that the results would not be confounded by problems with the language, we used experienced programmers familiar with the programming language. However, the programmers had no experience with the library — another similarity with realistic software development, in which new modules are often written within a relatively foreign context.

In essence, we used two independent variables: There were two separate problems to be solved (A and B, as described below) and two alternative treatments (ANSI C and K&R C, i.e., type checking and no type checking).

To balance for learning effects, sequence effects, and inter-subject ability differences, we used a counterbalanced design: Each subject had to solve both problems, each with a different language. The groups were balanced with respect to the order of both problem and language, giving a total of four experimental groups (see Table 1). Subjects were assigned to the groups randomly.

The design also allows to study a third independent variable, namely experience with the library: In his or her first task the subject has no previous experience while in the second task some experience from the first task is present.

The following subsections describe the tasks, the subjects, the experiment setup, and the observed variables and discuss internal and external validity of the experiment. Detailed information can be found in a technical report [15].

3.1 Tasks

Problem A (2×2 matrix inversion): Open a window with four text fields arranged in a 2×2 pattern plus an "Invert" and a "Quit" button. See Figure 1. "Quit" exits the program and closes the window. The text fields represent a matrix of real values. The values can be entered and edited. When the "Invert" button is pressed, replace the values by the coefficients of the corresponding inverted matrix, or print an error message if the matrix is not invertible. The formula for 2×2 matrix inversion was given.

Problem B (File Browser): Open a window with a menubar containing a single menu. The menu entry "Select file" opens a file-selector box. The entry "Open

Table 1: Tasks and compilers assigned to the four groups of subjects

	first problem A then problem B	first problem B then problem A
first ANSI C then K&R C	Group 1 8 subjects	Group 2 11 subjects
first K&R C then ANSI C	Group 3 8 subjects	Group 4 7 subjects

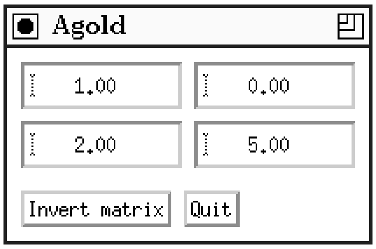

Figure 1: Problem A (2×2 matrix inversion)

selected file" pops up a separate, scrollable window and displays the contents of the file previously selected in the file selector box. "Quit" exits the program and closes all its windows. See Figure 2.

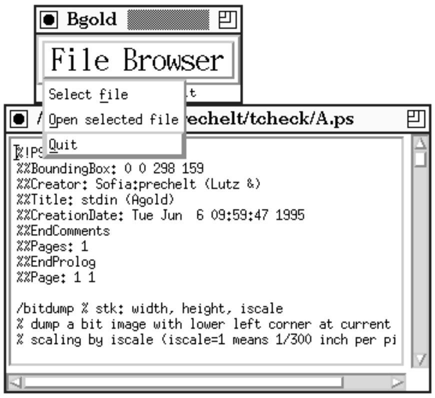

Figure 2: Problem B (File browser)

For solving the tasks, the subjects did not use native Motif, but a special wrapper library. The wrapper provides operations similar to those of Motif, but with improved type checking. For instance, all functions have fixed-length parameter lists, while Motif often provides variable-length parameter lists which are not checked. The wrapper also defines types for resource-name constants; in Motif, all resources are handled typelessly. Furthermore, the wrapper provides some simplification through additional convenience functions. For instance, there is a single function for creating a RowColumnManager and setting its orientation and packing mode; Motif requires several calls.

The tasks, although quite small, were not at all trivial. The subjects had to understand several important concepts of Motif programming (such as *widget*, *resource*, and *callback function*). Furthermore, they had to learn to use them from abstract documentation only, without example programs; we used no examples as we felt that these would have made the programming tasks too simple. Typically, the subjects took between one and two hours for their first task and about half that time for their second.

3.2 Subjects

A total of 40 unpaid volunteers participated in the study. Of those, 6 were removed from the sample: One deleted his protocol files, one was obviously too inexperienced (took almost 10 times as long as the others), and 4 worked on only one of the two problems. After this mortality, the A/B groups had 8+8 subjects and the B/A groups had 11+7 subjects. We consider this to be still sufficiently balanced [4].

The 34 subjects had the following education. 2 were postdocs in computer science (CS); 19 were PhD students in CS and had completed an MS degree in CS; another subject was also a CS PhD student but held an MS in physics; 12 subjects were CS graduate students with a BS in CS.

The subjects had between 4 and 19 years of programming experience ($\mu = 10.0$) and all but 11 of them had written at least 3000 lines in C (all but one at least 300 lines). Only 8 of the subjects had some programming experience with X-Windows or Motif; only 3 of them had written more than 300 lines in X-Windows or Motif.

3.3 Setup

Each subject received two written documents and one instruction sheet and was then left alone at a Sun-4 workstation to solve the two problems. The subjects were told to use roughly one hour per problem, but no time limit was enforced. Subjects could stop working even if the programs were not operational.

The instruction sheet was a one-page description of the global steps involved in the experiment: "Read sections 1 to 3 of the instruction document; fill in the questionnaire in section 2; initialize your working environment by typing `make TC1`; solve problem A by..." and so on. The subjects obtained the following materials, most of them both on paper and in files:

1. a half-page introduction to the purpose of the experiment
2. a questionnaire about the background of the subject
3. specifications of the two tasks plus the program skeleton for them
4. a short introduction to Motif programming (one page) and to some useful commands (for example to search manuals online)
5. a manual that listed first the names of all types, constants, and functions that might be required, followed by descriptions of each of them including the signature, semantic description, and several kinds of cross-references. The document also included introductions to the basic concepts of Motif and X-Windows. This manual was hand tailored to contain all information required to solve the tasks and hardly anything else.
6. a questionnaire about the experiment (to be filled in at the end)

Subjects could also execute a "gold" program for each task. The gold program solved its task completely and correctly and was to be used as a backup for the verbal specifications. Subjects were told to write programs that duplicated the behavior of the gold programs.

The subjects did not have to write the programs from scratch. Instead, they were given a program skeleton that contained all necessary `#include` commands, variable and function declarations, and some initialization statements. In addition, the skeleton contained pseudocode describing step by step what statements had to be inserted to complete the program. The subjects' task was to find out which functions they had to use and which arguments to supply. Almost all statements were function calls.

The following is an example of a pseudostatement in the skeleton.

```
/* Register callback-function 'button_pushed'
for the 'invert' button with the number 1 as
'client_data' */
```

It can be implemented thus:

```
XtAddCallbackF(invert, XmCactivateCallback,
button_pushed, (XtPointer)1);
```

There were only few variations possible in the implementation of the pseudocode.

The programming environment captured all program versions submitted for compilation along with a time stamp and the messages produced by the compiler and linker. A time stamp for the start and the end of the work phase for each problem was also written to the protocol file.

The environment was set up to call the standard C compiler of SunOS 4.1.3 using the command `cc -c -g` for the K&R tasks and version 2.7.0 of the GNU C compiler using `gcc -c -g -ansi -pedantic -W -Wimplicit -Wreturn-type` for the ANSI C tasks.

3.4 Dependent variables

For hypotheses 2 and 3 we observed when each individual defect in a program was introduced and removed. We also divided the defects in a few non-overlapping classes. We used the following procedure.

After the experiment was finished, each program version in the protocol files was annotated by hand. Each different defect that occurred in the programs was identified and given a unique number. For instance, for the call to `XtAddCallbackF` shown above, there were 15 different defect numbers, including 4 for wrong argument types, 4 for wrong argument objects with correct type, and another 7 for more specialized defects.

Each program version was annotated with the defects introduced, removed, or changed into another defect. Additional annotations counted the number of type defects, other semantic defects, and syntactic defects that actually provoked one or more error messages from the compiler or linker. The time stamps were corrected for work pauses that lasted more than 10 minutes in order to capture pure programming time only. Summary statistics were computed, for which each defect was classified into one of the following categories:

- *slight*: Defects resulting in slightly wrong functionality of the program, but so minor that a programmer may feel no need to correct them. Therefore, this class will also be ignored in order to avoid artifacts in the results.

- *invis*: Defects that are *invisible*, i.e., they do not compromise functionality, but only because of unspecified properties of the library implementation. Changes in the library implementation may result in a misbehaving program. Example: Supplying the integer constant `PACK_COLUMN` instead of the expected Boolean value `True` works correctly, because (and as long as) the constant happens to have a non-zero value. This rare class of defects will be ignored: *invis* defects can hardly be detected and thus are not relevant for our experiment.

- *invisD*: same as *invis*, except that the defects will be detected by ANSI C parameter type checking (but not by K&R C). The *invis* class excludes *invisD*.
- *severe*: Defects resulting in significant deviations from the prescribed functionality.
- *severeD*: same as *severe*, except that the defects will be detected by ANSI C parameter type checking (but not by K&R C). The *severe* class excludes *severeD*.

These categories are mutually exclusive. Defects that had to be removed before the program would pass even only the K&R C compiler and linker will be ignored. Unless otherwise noted, the defect statistics discussed below are computed based on the sum of *severe*, *severeD*, and *invisD*.

Other metrics observed were the number of compilation cycles (versions) and time to delivery, i.e., the time spent by the subjects before delivering the program (whether complete and correct or not).

From these metrics and annotations, additional statistics were computed. For instance the frequency of defect insertion and removal, the number of attempts made before a defect was finally removed, the Interface Defect Lifetime, and the number and type of defects remaining in the final program version. See also the definitions in Section 1.

For measuring productivity and unimplemented functionality, we define a *functionality unit (FU)* to be a single statement in the gold program. For example, the call to `XtAddCallbackF` shown in Section 3.3 is one FU. Using the gold programs as a reference normalizes the cases in which subjects produce more than one statement instead. FUs are thus a better measure of program volume than lines of code. Gold program A contains 16 FUs, B contains 11. We annotated the programs with the number of *gaps*, i.e., the number of missing FUs. An FU is counted as missing if a subject made no attempt to implement it.

3.5 Internal and external validity

The following problems might threaten the internal validity of the experiment, i.e., the correctness of the results:

1. For defects where both the K&R and the ANSI C compiler produce an error message, these messages might differ and this might influence productivity. Our subjective judgment here is that for the purposes of this experiment the error messages of both compilers, although sometimes quite different, are overall comparable in quality. Furthermore, none of our subjects were very experienced with one particular compiler and would understand its messages faster than others.
2. There may be annotation errors. To insure consistency, all annotations were made by the same person. The annotations were cross-checked first with a simple consistency checker (looking whether errors were introduced before removed, times were plausible, etc.), and then some of them were checked manually. The number of annotation mistakes found in the manual check was negligible (about 4%).
3. The learning effect from first to second task might be different for K&R subjects than for ANSI C subjects. This problem, and related ones, is accounted for by the counter-balanced experiment design.

The following problems might limit external validity of the experiment, i.e., the generalizability of our results:

1. The subjects were not professional software engineers. However, they were quite experienced programmers and held degrees (many of them advanced) in computer science.
2. The results may be domain dependent. This objection cannot be ruled out. This experiment should therefore be repeated in domains other than graphical user interfaces.
3. The results may or may not apply to situations in which the subjects are very familiar with the interfaces used. This question might also be worth a separate experiment.

Despite these problems, we believe that the scenario chosen in the experiment is nevertheless similar to many real situations with respect to type-checking errors.

Another issue is worth discussing here: The learning effect (performance change from first task to second task) is larger than the treatment effect (performance change from K&R C to ANSI C). This would be a problem if the learning reduced the treatment effect [16, pages 106 and 113]. However, as we will see below, in our case the treatment effect is actually *increased* by learning, making our experiment results conservative ones. We are explicitly considering programmers who are not highly familiar with the interface used. Therefore learning is a natural and necessary part of our setting, not an artifact of improper subject selection.

4 Results and Discussion

Many of the statistics of interest in this study have clearly non-normal distributions and sometimes severe outliers. Therefore, we present medians (to be precise: an interpolated 50% quantile) rather than arithmetic means. Where most of the median values are zero, higher quantiles are given.

The results are shown in Tables 2 through 4. There are altogether ten different statistics, each appearing in three main columns. The first column shows the statistics for both tasks, independent of order. The second and third columns reflect the observations for those tasks that were tackled first and second, respectively. These columns can be used to assess the learning effect. Each main column reports the medians (or higher quantiles where indicated) for the tasks programmed with ANSI C and K&R C plus the p-value. The p-value is the result of the Wilcoxon Rank Sum Test (Mann-Whitney U Test) and, very roughly speaking, represents the probability (given the observations) that the medians of the two samples are equal. If $p \leq 0.05$, the test result is considered statistically significant and we call the distributions significantly different. Significant results are marked in boldface in the tables. When the result is not significant, nothing can be said; there may or may not be a difference.

4.1 Productivity

Table 2 shows three measures that describe the overall time taken and the productivity exhibited by the subjects.

Statistic 1, time to delivery, shows no significant difference between ANSI C and K&R C for the first task or for both tasks taken together. Ignoring the programming language, the time spent for the second task is shorter than for the first ($p = 0.0012$, not shown in the table), indicating a learning effect. In the second task, ANSI C programs are delivered significantly faster than K&R C programs. A plausible explanation is that when they started, programmers did not have a good understanding of the library and were struggling more with the concepts than with the interface itself. This explanation was confirmed by studying the compiler inputs. Type checking is unlikely to help gain a better understanding. Type checks became useful only after programmers had overcome the initial learning hurdle.

Statistic 2, the number of program versions compiled, does not show a significant difference; ANSI C programmers compile about as often as K&R C programmers.

Statistic 3 describes the productivity measured in functional units per hour (FU/h). In contrast to time to delivery, this value accounts for functionality not implemented by a few of the subjects. Again we find no significant difference for the first task, but a (weakly) significant difference for the second task. There, ANSI C median productivity is about 20% higher than K&R C productivity, suggesting that ANSI C is helpful for programmers after the initial interface learning phase. This observation supports hypothesis 1. The combined (both languages) productivity rises very significantly from the first task to the second task ($p = 0.0001$, not shown in the table); this was also reported by the subjects and confirms that there is a strong learning effect induced by the sequence of tasks. The actual distributions of productivity measured in FU/h are shown in Figures 3 to 5. We see that ANSI C makes for a more pronounced increase in productivity from the first task to the second (about 78% for the median) than does K&R C (about 26% for the median).

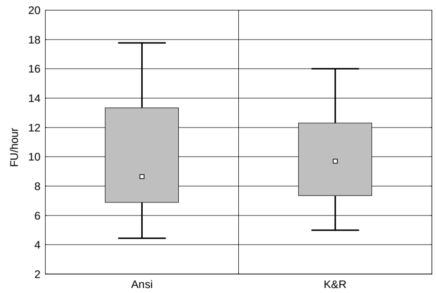

Figure 3: Boxplots of productivity (in FU/hour) over both tasks for ANSI C (left boxplot) and K&R C (right boxplot). The upper and lower whiskers mark the 95% and 5% quantiles, the upper and lower edges of the box mark the 75% and 25% quantiles, and the dot marks the 50% quantile (median). All other boxplots following below have the same structure.

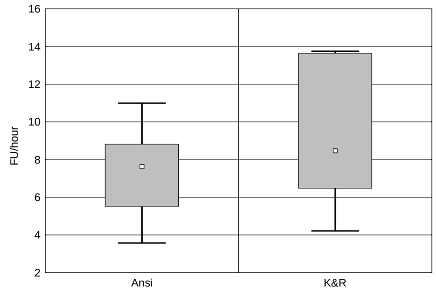

Figure 4: Boxplots of productivity (in FU/hour) for first task.

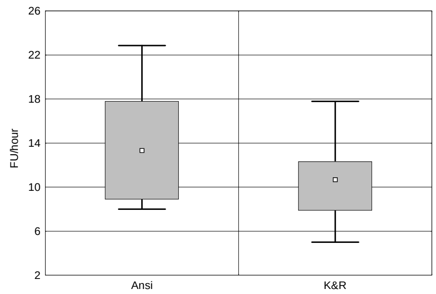

Figure 5: Boxplots of productivity (in FU/hour) for second task.

Table 2: Overall productivity statistics. Medians of statistics for Ansi C vs. K&R C versions of programs and p-values for statistical significance of Wilcoxon Rank Sum Tests of the two. Values under 0.05 indicate significant differences of the medians. Column pairs are for 1st+2nd, 1st, and 2nd problem tackled chronologically by each subject, respectively. All entries include data points for both problem A and problem B.

	Statistic	both tasks		1st task		2nd task	
		Ansi	K&R	Ansi	K&R	Ansi	K&R
1	hours to delivery	1.3	1.35	1.6	1.6	0.9	1.3
	$p =$	*0.49*		*0.83*		**0.018**	
2	#versions	15	16	19	21	12.5	13
	$p =$	*0.84*		*0.63*		*0.16*	
3	FU/h	8.6	9.7	7.2	8.5	12.8	10.7
	$p =$	*0.93*		*0.31*		*0.061*	

Table 3: Statistics on internals of the programming process. See Table 2 for explanations.

	Statistic	both tasks		1st task		2nd task	
		Ansi	K&R	Ansi	K&R	Ansi	K&R
4	accumul. interf. dfct. lifetime (median)	0.3	1.2	0.5	2.1	0.2	1.1
	$p =$	**0.004**		**0.028**		*0.059*	
5	#right, then wrong again (75% quant.)	1.0	1.0	1.0	1.0	0.0	1.0
	$p =$	*0.12*		*0.82*		**0.009**	

4.2 Defect lifetimes

Table 3 gives some insight into the programming process.

Statistic 4 is the time from the introduction of an interface defect to its removal (or the end of the experiment) accumulated over all interface defects introduced by a subject. The distributions of this variable over both tasks are also shown as boxplots in Figure 6. As

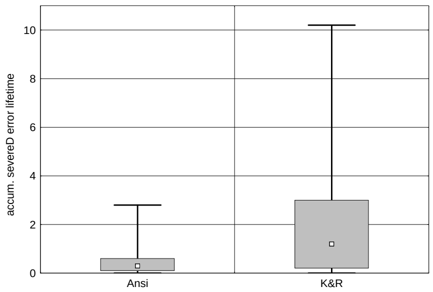

Figure 6: Boxplots of accumulated interface defect lifetime (in hours) over both tasks.

we see, the K&R total defect lifetimes are higher and spread over a much wider range; the difference is significant. Note that the frequency of defect insertion (number of interface defects inserted per hour, not shown in the table) does not show significant differences between the languages, indicating that Ansi C is of little help in defect prevention (as opposed to defect removal). Taken together, these two facts support hypothesis 3: Ansi C helps to remove interface defects quickly.

Statistic 5 indicates the number of defects, interface or other, introduced in previously correct or repaired statements of a program. While there is hardly any difference in the first task, the value is significantly higher for K&R C in the second task. We speculate that this happens because the type error messages of Ansi C allow some of the subjects to avoid the trial-and-error defect removal techniques they would have used in K&R C; the effect occurs only in the second task, after the subjects have gained a basic understanding of Motif concepts.

4.3 Defects in delivered programs

Table 4 describes the quality of the products delivered by the subjects.

Statistic 6 says that there are *not* more unimplemented functionality units ("gaps") in the K&R C programs.

Statistic 7 confirms that there are more defects in the delivered K&R C programs than in the Ansi C programs; see also the distribution as shown in Figure 7. The difference is much more pronounced in the second task, though. Again the reason is probably that the advantages of Ansi C become fully relevant only after most other initial problems have been mastered. Statistics 8 to 10 confirm that the reason for the difference lies indeed in the type checking capabilities of Ansi C: both the rare *invisD* defects (statistic 8) and the *severeD* defects (statistic 10, see also Figure 8) are much less frequent in delivered Ansi C programs than in K&R C programs. These defects can be detected by Ansi C type checking. On the other hand, *severe* defects (statistic 9, see also Figure 9) are about as frequent in delivered Ansi C programs as in K&R C programs. These defects cannot be detected by type checking.

As we see in the boxplots, the distributions for *severe*

Table 4: Statistics on the delivered program. See Table 2 for explanations. Lines 6 and 8 do not list medians but other quantiles instead, as indicated.

	Statistic	both tasks		1st task		2nd task	
		ANSI	K&R	ANSI	K&R	ANSI	K&R
6	#gaps (75% quantile)	0.25	0.0	1.5	0.0	0.0	0.0
	$p =$	*0.35*		*0.26*		*0.70*	
7	#remaining errs in delivered program	1.0	2.0	1.0	2.0	1.0	2.0
	$p =$	**0.016**		*0.32*		**0.031**	
8	— for *invisD* only (90% quantile)	0.0	1.0	0.0	1.4	0.0	0.0
	$p =$	**0.04**		**0.048**		*0.41*	
9	— for *severe* only	1.0	1.0	1.0	0.0	1.0	1.0
	$p =$	*0.66*		*0.74*		*0.65*	
10	— for *severeD* only	0.0	1.0	0.0	1.0	0.0	1.0
	$p =$	**0.0001**		**0.015**		**0.0022**	

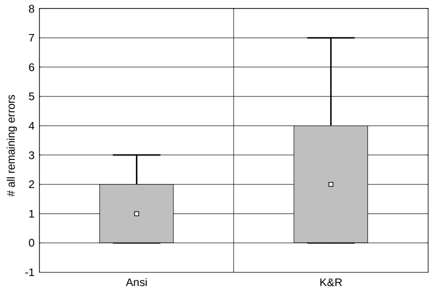

Figure 7: Boxplots of total number of remaining defects in delivered programs over both tasks.

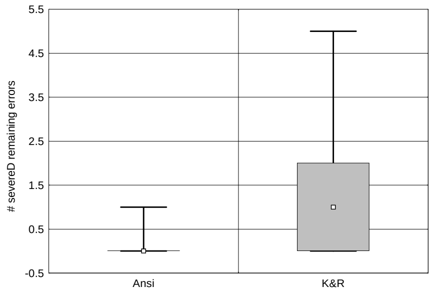

Figure 8: Boxplots of number of remaining *severeD* defects in delivered programs over both tasks.

defects differ only in the upper tail, whereas the distributions for the *severeD* defects differ dramatically in favor of ANSI C, resulting in a significant overall advantage for ANSI C. These observations support hypothesis 2.

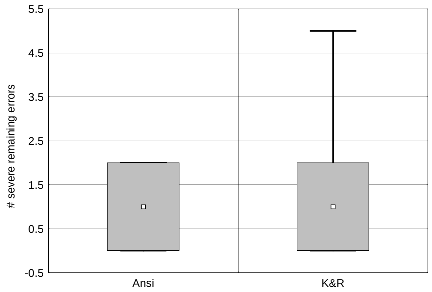

Figure 9: Boxplots of number of remaining *severe* defects in delivered programs over both tasks.

Detailed analysis of the defects remaining in the delivered programs indicates a slight, but not statistically significant tendency that besides type defects other classes of frequent defects also were reduced in the ANSI C programs: using the wrong variable as a parameter or an assignment target ($p = 0.28$) or using a wrong constant value as a parameter ($p = 0.35$). It is unknown whether this is a systematic side-effect of type checking and how it should be explained if it is.

There were no significant differences between the two tasks; all of the above results hardly change if one considers the tasks A and B separately.

4.4 Questionnaire results

Finally, the subjective impressions of the subjects as reported in the questionnaires are as follows: 26 of the subjects (79%) noted a learning effect from the first program to the second. 9 subjects (27%) reported that they found the ANSI C type checking very helpful, 11 (33%) found it considerably helpful, 4 (12%) found it almost not helpful, 5 (15%) found it not at all helpful. 4 subjects could not decide and 1 questionnaire was lost.

5 Conclusions and further work

The experiment results allow for the following statements regarding our hypotheses:

- **Hypothesis 1, Interface Use Productivity:** When programming an interface, type checking increases productivity, provided the programmer has gained a basic understanding of the interface.

- **Hypothesis 2, Interface Defects in delivered program:** Type checking reduces the number of Interface Defects in delivered programs.

- **Hypothesis 3, Interface Defect Lifetime:** Type checking reduces the time defects stay in the program during development.

One must be careful generalizing the results of this study to other situations. For instance, the experiment is unsuitable for determining the proportion of interface defects in an overall mix of defects, because it was designed to prevent errors other than interface errors. Hence it is unclear how large the differences will be if defect classes such as declaration defects, initialization defects, algorithmic defects, or control-flow defects are included.

Nevertheless, the experiment suggests that for many realistic programming tasks, type checking of interfaces improves both productivity and program quality. Furthermore, some of the resources otherwise expended on inspecting interfaces might be allocated to other tasks. As a corollary, library design should strive to maximize the type-checkability of the interfaces by introducing new types instead of using standard types where appropriate. For instance Motif, on which our experiment library was based, is a negative example in this respect.

Further work should repeat similar error and defect analyses in different settings (e.g. tasks with complex data flow or object-oriented languages). In particular, it would be interesting to compare productivity and error rates under compile-time type checking, runtime type checking, and type inference. Other important questions concern the influence of a disciplined programming process such as the Personal Software Process [12]. Finally, an analysis of the errors occurring in practice might help devise more effective defect-detection mechanisms.

Acknowledgments

We thank Paul Lukowicz for patiently guinea-pigging the experimental setup, Dennis Goldenson for his detailed comments on an early draft, and Larry Votta for pointing out an important reference and providing many suggestions on the report. Last, but not least, we thank our subjects.

A Solution for Problem A

This is the program (ANSI C version) that represents the canonical solution for Problem A. Most of it, including all of the comments, was given to the subjects from the start; they only had to insert the statements marked here with `/*FU 1*/` etc. at those places previously held by pseudocode comments as described in Section 3.3 above. The numbers in the `FU` comments count the functional units as defined in Section 1.

```
#include <stdio.h>
#include <stdlib.h>
#include "stdmotif.h"

void button_pushed (Widget widget, XtPointer client_data, XtPointer call_data);
Widget mw[4];  /* text fields for matrix coefficients: 0,1,2,3 for a,b,c,d */

/************************ MAIN PROGRAM *************************/
int main (argc, argv)
  int argc;
  char *argv[];
{
  Widget        toplevel, /* main window */
                manager,  /* manager for square and buttons */
                square,   /* manager for 4 TextFields */
                buttons,  /* manager for 2 PushButtons */
                invert,   /* PushButton */
                quit;     /* PushButton */
  XtAppContext  app;
  XmString  invertlabel, quitlabel;

  /*---------- 1. initialize X and Motif -------------------------------*/
  /* (already complete, should not be changed) */
  globalInitialize ("A");
  toplevel = XtVaAppInitialize (&app, "Hello", NULL, 0,
        &argc, argv, fallbacks, NULL);

  /*---------- 2. create and configure widgets -------------------------*/
  manager = XmCreateRowColumnManagerOCP ("manager", toplevel, XmVERTICAL,
                                         2, False);           /*FU 1*/
  square = XmCreateRowColumnManagerOCP ("square", manager, XmHORIZONTAL,
                                        2, True);             /*FU 2*/
  buttons = XmCreateRowColumnManagerOCP ("buttons", manager, XmHORIZONTAL,
                                         1, False);           /*FU 3*/
  mw[0] = XmCreateTextFieldWidgetW ("aw", square, 100, "a");  /*FU 4*/
  mw[1] = XmCreateTextFieldWidgetW ("bw", square, 100, "b");  /*FU 5*/
  mw[2] = XmCreateTextFieldWidgetW ("cw", square, 100, "c");  /*FU 6*/
  mw[3] = XmCreateTextFieldWidgetW ("dw", square, 100, "d");  /*FU 7*/
  invert = XmCreatePushButtonL ("invert", buttons,            /*FU 8*/
                           XmStringCreateLocalized ("Invert matrix"));
  quit = XmCreatePushButtonL ("quit", buttons,                /*FU 9*/
                           XmStringCreateLocalized ("Quit"));

  /*---------- 3. register callback functions --------------------------*/
  XtAddCallbackF (invert, XmCactivateCallback, button_pushed,
                  (XtPointer)1);                              /*FU 10*/
  XtAddCallbackF (quit, XmCactivateCallback, button_pushed,
                  (XtPointer)99);                             /*FU 11*/

  /*---------- 4. realize widgets and turn control to X event loop ------*/
  /* (already complete, should not be changed) */
  XtRealizeWidget (toplevel);
  XtAppMainLoop (app);
  return (0);
```

```
}

/************************ Functions ************************/
void button_pushed (Widget widget, XtPointer client_data, XtPointer call_data)
{
  /* this is the callback function to be called when clicking on
     the PushButtons occurs */
  double mat[4], new[4], /* old and new matrix coefficients */
         det;            /* determinant */
  String s;
  if ((int)client_data == 99)
     exit (0);                                          /*FU 12*/
  else if ((int)client_data == 1) {
    int i;
    for (i = 0; i <= 3; i++) {
      XtGetStringValue (mw[i], XmCvalue, &s);           /*FU 13*/
      mat[i] = atof (s);                                /*FU 14*/
    }
    det = mat[0]*mat[3] - mat[1]*mat[2];
    if (det != 0) {
      new[0] = mat[3]/det; new[1] = -mat[1]/det;
      new[2] = -mat[2]/det; new[3] = mat[0]/det;
      for (i = 0; i <= 3; i++)
        XtSetStringValue (mw[i], XmCvalue, ftoa(new[i],8,2)); /*FU 15*/
    }
    else
      matrixErrorMessage("Matrix cannot be inverted",mat,8,2);/*FU 16*/
  }
}
```

B Solution for Problem B

See the description in Appendix A above.

```
#include <stdio.h>
#include <stdlib.h>
#include "stdmotif.h"

void handle_menu (Widget widget, XtPointer client_data, XtPointer call_data);

Widget toplevel;

/************************ MAIN PROGRAM *************************/
int main (int argc, char *argv[])
{
  Widget        main_w,  /* main window */
                menubar, /* the one-entry menu bar */
                menu,    /* the pulldown menu */
                label;   /* the label displayed in the work window */
  XtAppContext  app;

  /*---------- 1. initialize X and Motif -----------------------------*/
  /* (already complete, should not be changed) */
  globalInitialize ("B");
  toplevel = XtVaAppInitialize (&app, "Hello", NULL, 0,
        &argc, argv, fallbacks, NULL);

  /*---------- 2. create and configure widgets -----------------------*/
  main_w = XmCreateMainWindowWidget ("main_window", toplevel);/*FU 1*/
```

```
  menubar = XmCreateTrivialMenuBar (main_w, "FileBrowser",
              XmStringCreate ("File Browser", "LARGE"), 'F'); /*FU 2*/
  label = XmCreateLabelWidget ("by", main_w,
              XmStringCreate ("by Lutz Prechelt", "SMALL"));  /*FU 3*/
  XtSetWidgetValue (main_w, XmCworkWindow, label);            /*FU 4*/
  menu = XmCreatePulldownMenu3 ("TheMenu", menubar, 0,
                  XmStringCreateLocalized ("Select file"), 'f',
                  XmStringCreateLocalized ("Open selected file"), 'O',
                  XmStringCreateLocalized ("Quit"), 'Q',
                  handle_menu);                               /*FU 5*/

  /*---------- 3. register callback functions -------------------------*/
  /* (handle_menu was already registered above, nothing to be done) */

  /*---------- 4. realize widgets and turn control to X event loop ----*/
  /* (already complete, should not be changed) */
  XtRealizeWidget (toplevel);
  XtAppMainLoop (app);
  return (0);
}

/************************ Functions ************************/

void handle_menu (Widget widget, XtPointer client_data,
                  XtPointer call_data)
{
  if ((int)client_data == 0) {  /* first menu entry selected */
    Widget fs = XmCreateFileSelectorDialog (toplevel,
                                        "fileselection"); /*FU 6*/
    XtAddCallbackF (fs, XmCokCallback, keepSelectedFile, NULL); /*FU 7*/
    XtManageChild (fs);                                       /*FU 8*/
  }
  else if ((int)client_data == 1) {  /* second menu entry selected */
    Widget scrolltext = XmCreateScrolledTextWindow (selectedFile(),
                             toplevel, 25, 80);               /*FU 9*/
    XtSetStringValue (scrolltext, XmCvalue,
                      readWholeFile (selectedFile()));        /*FU 10*/
  }
  else if ((int)client_data == 2) {  /* third menu entry selected */
    exit (0);                                                 /*FU 11*/
}
```

References

[1] Victor R. Basili and B.T. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 27(1):42–52, January 1984.

[2] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

[3] Kim Bruce. Typing in object-oriented languages: Achieving expressibility and safety. *ACM Computing Surveys?*, .(.):., 1998? to appear, see http://www.cs.williams.edu/˜kim/.

[4] Larry B. Christensen. *Experimental Methodology.* Allyn and Bacon, Needham Heights, MA, 6th edition, 1994.

[5] Curtis R. Cook, Jean C. Scholtz, and James C. Spohrer, editors. *Empirical Studies of Programmers: Fifth Workshop*, Palo Alto, CA, December 1993. Ablex Publishing Corp.

[6] Alireza Ebrahimi. Novice programmer errors: Language constructs and plan composition. *Intl. J. of Human-Computer Studies*, 41:457–480, 1994.

[7] Marc Eisenstadt. Tales of debugging from the front lines. In *[5]*, pages 86–112, 1993.

[8] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. on Software Engineering*, 1993.

[9] J.D. Gannon. An experimental evaluation of data type conventions. *Communications of the ACM*, 1977.

[10] Robert B. Grady. Practical results from measuring software quality. *Communications of the ACM*, 36(11):62–68, November 1993.

[11] Paul Hudak and Mark P. Jones. Haskell vs. Ada vs. C++ vs. awk vs. ... an experiment in software prototyping productivity. Technical report, Yale University, Dept. of CS, New Haven, CT, July 1994.

[12] Watts Humphrey. *A Discipline for Software Engineering.* SEI Series in Software Engineering. Addison Wesley, Reading, MA, 1995.

[13] Murthi Nanja and Curtis R. Cook. An analysis of the on-line debugging process. In *[14]*, pages 172–184, 1987.

[14] Gary M. Olson, Sylvia Sheppard, and Elliot Soloway, editors. *Empirical Studies of Programmers: Second Workshop*, Washington, D.C., December 1987. Ablex Publishing Corp.

[15] Lutz Prechelt and Walter F. Tichy. A controlled experiment measuring the impact of procedure argument type checking on programmer productivity. Technical Report CMU/SEI-96-TR-014, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, June 1996.

[16] B.A. Sheil. The psychological study of programming. *ACM Computing Surveys*, 1981.

[17] Elliot Soloway and Sitharama Iyengar, editors. *Empirical Studies of Programmers.* Ablex Publishing Corp., Norwood, NJ, June 1986. (The papers of the First Workshop on Empirical Studies of Programmers, Washington D.C.).

[18] James G. Spohrer and Elliot Soloway. Analyzing the high frequency bugs in novice programs. In *[17]*, pages 230–251, 1986.

[19] Webb Stacy and Jean MacMillian. Cognitive bias in software engineering. *Communications of the ACM*, 38(6):57–63, June 1995.

[20] Barbee Teasley, Laura Marie Leventhal, and Diane S. Rohlman. Positive test bias in software testing by professionals: what's right and what's wrong. In *Empirical Studies of Programmers: Fifth Workshop*, pages 206–221, Palo Alto, CA, December 1993. Ablex Publishing Corp.

[21] Nikolaus Wirth. Gedanken zur Software-Explosion. *Informatik Spektrum*, 17(1):5–20, February 1994.

[22] Claes Wohlin and Per Runeson. Certification of software components. *IEEE Trans. on Software Engineering*, 20(6):494–499, June 1994.

Biographical sketch

Lutz Prechelt works as senior researcher at the Informatics department of the University of Karlsruhe. There he also received his diploma (1990) and his Ph.D (1995) in Informatics. His research interests include empirical software engineering, software design patterns, compiler construction for parallel machines, constructive neural network learning algorithms, measurement and benchmarking issues, and research methodology. Prechelt is a member of IEEE, ACM, and GI.

Walter F. Tichy is professor of Computer Science at the University Karlsruhe, Germany. Previously, he was senior scientist at Carnegie Group, Inc., in Pittsburgh, Pennsylvania and on the faculty of Computer Science at Purdue University in West Lafayette, Indiana. His primary research interests are software engineering and parallelism. He is currently involved in a number of projects, including experimental methods in computer science, software architecture and design patterns, software configuration management, workstation clusters, optoelectronic interconnects for parallel computers, and optimizing compilers for parallel computers. He has consulted widely for industry.

Tichy received a B.S. from the Technical University in Munich in 1974 and M.S. and Ph.D. degrees in Computer Science from Carnegie-Mellon University in 1976 and 1980. He is a member of ACM, GI, and IEEE Computer Society.