The software quality and $\mathcal{NTS}$

Karel Skoupý
Tábor 32A, 602 00 Brno, Czechia
skoupy@fi.muni.cz
http://www.fi.muni.cz/~skoupy

Abstract

$\mathcal{NTS}$ is a project which aimed to come up with a successor of TeX. The first step was a modular reimplementation of TeX in order to pave a way for changes, extensions and experiments. This article evaluates the current state of $\mathcal{NTS}$ in terms of software quality factors and discusses if the purpose of the project has been achieved and what changes should be made in the future development.

Introduction

$\mathcal{NTS}$ is a project which aimed to come up with a successor of TeX. The first step of the chosen strategy was a reimplementation of the "old good" TeX in a modular way in order to pave the way for changes, extensions and experiments. To achieve the goal of extendibility the *Object Oriented* approach has been chosen and Java became the language of implementation. The reimplementation is not a simple port but an original (re)design. The original TeX code has not been reused and at the same time the requirement that the $\mathcal{NTS}$ ffunctionality be identical as that of TeX has been observed.

A few articles on $\mathcal{NTS}$ have already been published in *TUGboat* and proceedings of several TeX Users Group conferences. [6] describes the history and conception, [3] discusses the future goals and possibilities and [10, 11] explain the design of the reimplemented system and the roles of key classes and packages.

As the first (or zeroth) version of $\mathcal{NTS}$ is almost finished it is time to evaluate some of the aspects of the resulting program. As guidelines for such evaluation I borrowed a quite interesting list of external software quality factors (including their definitions) published in [7].

Evaluation

The external quality factors (how the software behaves) are important to users. There are also internal factors (how the software is written) which are interesting to developers. In the end only the external factors matter (you don't care how beautiful the code is which produced the incorrect and slow output) although the internal ones are crucial for achieving the desired external characteristics.

Because $\mathcal{NTS}$ is strongly connected to the original TeX it comes quite naturally that comments about these quality factors in TeX will also be made.

Correctness *Correctness is the ability of software products to perform their exact tasks, as defined by their specification.*[1]

The key point in the discussion of this factor is the specification. The specification of $\mathcal{NTS}$ defined by the project is "simple". It must be as much as possible (99.99%) compatible with TeX in the sense that it should give the same DVI, log file and screen outputs for the same input files as the original TeX does. Exceptions from this rule are allowed only if they are well understood and there is a good reason for them. TeX is for the needs of the project correct by definition.

Although the specification above is on one hand extremely short, on the other hand it is quite difficult and complicated. There exists no formal specification of what TeX exactly does. *The TeXbook* was meant rather to be a user manual and as a specification it is insufficient and sometimes even confusing. Maybe you already had to go from dedicated chapters through "Summary of Vertical Mode" to "Dirty Tricks" to find an important detail. And definitely one cannot learn the precise formats of the log or DVI outputs from it.

The only precise "specification" is therefore *TeX: The Program* itself. But even this was not enough for the purpose of testing and comparing actual outputs of TeX and $\mathcal{NTS}$. Some small parts of *TeX: The Program* are being regarded as "system dependent" and are left open for concrete ports. For pragmatic reasons I have adopted the web2c implementation available with my Linux system as the "canonical" TeX.

[1] [7] p. 4

In the course of the reimplementation I had to go through and understand each detail of the `tex.web` as well as parts of `tex.ch` and `web2c/lib` code to make sure that the outputs from NTS and TEX become exactly the same. Later during the testing of more difficult or obscure cases (as the reconstitution of ligatures and kernings after hyphenation or the converting of math lists into boxes) I also had to insert some tracing code into TEX to really understand what is going on there.

In spite of my best efforts the 100% functional identity of NTS with TEX will not be achieved. One of the elements which will not be simulated is TEX's memory usage statistics. As the data structures in NTS are completely different it obviously makes no sense to mimic the corresponding TEX outputs. Other areas are much more obscure and you will probably never encounter them. Some of them can show up in the TRIPTEST. In general these areas are related to the log output—the cost of mimicking the behaviour of TEX would be too high (substantial changes in important object interfaces) and not worth the result. There should be no differences in the binary DVI output (except for very rare rounding errors which are due to differences in the floating point arithmetics which also may occur in TEX on different machines).

It is infeasible to prove formally the correctness of NTS (one also would need a formal specification of TEX in the first place). The only means for ensuring that NTS is *probably* correct is by thorough testing. The requirement for identical functionality with TEX is interestingly enough very helpful here. Testing itself is relatively simple. The testing documents are being processed by both TEX and NTS and DVI and log outputs are compared. In most of the cases with the exceptions mentioned above the outputs should be identical.

At the time of writing this article NTS successfully processes *The TEXbook*, *The METAFONTbook* and people have reported the successful processing of other complicated documents written in LATEX or ConTEXt. There are still some bugs but I'm working to iron them out. Currently about half of the TRIPTEST is being processed successfully.

Robustness *Robustness is the ability of software systems to react appropriately to abnormal conditions.*[2]

In the case of typesetting this requirement is the ability to handle incorrect input, missing font metrics, unexpected user input, etc. The robustness of NTS is closely linked to the robustness of TEX itself because the programs are functionally (almost) identical for all inputs.

NTS exactly follows the TEX behaviour when reacting to incorrect input. It prints (sometimes confusing) error messages, tries to recover by inserting/deleting tokens, allows the user to correct mistakes (in interactive modes) and provides hints. Sometimes it quits prematurely but decently in reaction to fatal errors like "interwoven alignment templates are not allowed".

In some cases TEX goes into an infinite loop. One such case is an unbalanced output routine when reading from a file (it probably does not occur too often). The behaviour of NTS in this case is not decided yet (still on the TODO list) but I hope that people will not get angry if it is not fully compatible with TEX.

There might be even cases when TEX crashes but I am not aware of them. If some TEXperts could provide me with such problematic inputs it would be very interesting to see how NTS handles them.

Another important robustness issue is the implementation language: Java. The pending Java problem is that there is no defined way to handle signals (interrupts). In other words if the user (by accident) presses `control-C` in the middle of the job the process is simply killed. TEX in such a case prompts the user as if it were a (synchronous) input error. So far I have not found an acceptable solution. I only discovered some Sun proprietary classes which handle that but they are platform dependent, undocumented and their use is being discouraged. I wonder how Java can be used for more critical tasks with such a deficiency. If somebody knows what should be done, please, do let me know.

Extendibility *Extendibility is the ease of adapting software products to changes of specification.*[3]

During its lifetime extendibility became a week spot of TEX. As a matter of fact it became the main reason for starting the NTS project. This section is not being intended to be a critique of the TEX design or code. TEX has always served and amazingly enough still serves its purpose very well. But it is the extendibility which although great compared to other programs of that time that has simply been stretched to the limits by such recent developments like e.g. pdfTEX.

An important design principle which helps to achieve extendibility is modularity (see [7] for rationale). The full definition of modularity is rather complex but let us say that a modular system is

[2] [7] p. 5

[3] [7] p. 6

a system which can be decomposed into pieces (modules) that are self-contained, have clearly defined interfaces and are organised into a stable architecture.

Although *TEX: The Program* uses the word "module" for its building blocks and the facilities provided by `WEB`[4] were a significant step towards modularity, these modules are neither self-contained nor have clearly defined interfaces. References to both local and global variables as well as labels (`goto` targets) cross the module boundaries. Most frequently used data structures are so highly optimised in their memory requirements that they are not type safe and are very difficult to change. There are probably two reasons for this: lack of support in programming technologies and methodologies and low speed and memory limitations of computers in those days.

Modularity was the first objective for $\mathcal{NTS}$. I have been trying very hard to decompose the program into autonomous pieces on two levels. One was the level of packages which divides the whole system into autonomous subareas of macro language, typesetting, processing of math formulas, hyphenation, font metric files, alignment, producing DVI and several others. The second level is the level of classes. A separate class has been defined for each important concept of TEX and many subclasses for various specialisations of those concepts. The relations between classes have been defined clearly and explicitly. One of the $\mathcal{NTS}$ alpha reviewers even reported that the design is over-modularised. For more details about $\mathcal{NTS}$ classes and packages see [10, 11].

The requirement that $\mathcal{NTS}$ behaves in exactly the same way as TEX unfortunately did often lead to complications. The TEX code is sometimes slightly inconsistent in the sense that many algorithms are conceptually very similar but the code is repeated with small differences. An example may be the packaging of hboxes and vboxes, breaking of a page or splitting a vertical list or such minor things like printing inserted token lists of different kinds in an error context trace in the log file. In $\mathcal{NTS}$ similar concepts are unified but then rather artificial methods or parameters must be introduced to accommodate those inconsistencies. The TEX code lacks protection of data. Direct access to some distant data structures is often very efficient and even elegant but creates many dependencies. These dependencies and inconsistencies are sometimes unfortunately not just internal issues but are reflected in the external behaviour. TEX reveals too many details about its internal data and ways of handling them in its log outputs. Look for example at the output of `\showlists` inside a denominator of fraction inside a displayed formula. Although $\mathcal{NTS}$ tries to isolate all dependencies and to express them in clean interfaces the result is sometimes rather complicated and may lead to decreased ease of future extensions implementation.

The modularity of the design and implementation of $\mathcal{NTS}$ should prove a sound basis for extendibility although I have to admit that this aspect of $\mathcal{NTS}$ has not been tested yet.

Reusability *Reusability is the ability of software elements to serve for the construction of many different applications.*[5]

Isolating important parts of TEX in such a way that they can be used in other applications was one of the major project objectives. There were proposals to write a system with a graphical user interface using the best fit paragraph builder or providing a browser plug-in for online typesetting.

In the times when TEX was born the code reuse was done mostly by cutting and pasting. A good example are the TEX and `tftopl` programs. Both use the same file format, a lot of code is similar but nothing is shared. `tftopl` has actually been reimplemented in Java—it has only about 40 lines of code because it uses the class which in $\mathcal{NTS}$ encapsulates the TFM format.

Reusability is tightly coupled with extendibility and both factors are supported by modularity. It is important to note that the components to be reused in other applications should be independent. A lot of effort has been put especially into the elimination of cyclic dependencies. The packages and classes are ordered into hierarchies which ensure that lower levels do not depend on higher ones (see Figure 1). This facilitates the isolation of selected subsets of $\mathcal{NTS}$ packages for reuse in other applications. More detailed discussion of this principle can be found in [10].

Unfortunately it turned out that Java, the language of implementation, does not support the modularity to the extent to be found in other contemporary languages. Two important features supporting modularity are missing: multiple inheritance and genericity[6]. Multiple inheritance is actually present but only for interfaces and not for classes. This leads to code duplication in classes which are conceptual specialisations of more than one ancestor. Genericity is more important because it not only allows to

[4] A preprocessor facilitating literate programming, by D. E. Knuth.

[5] [7] p. 7

[6] Parametrisation of types and functions by other types.

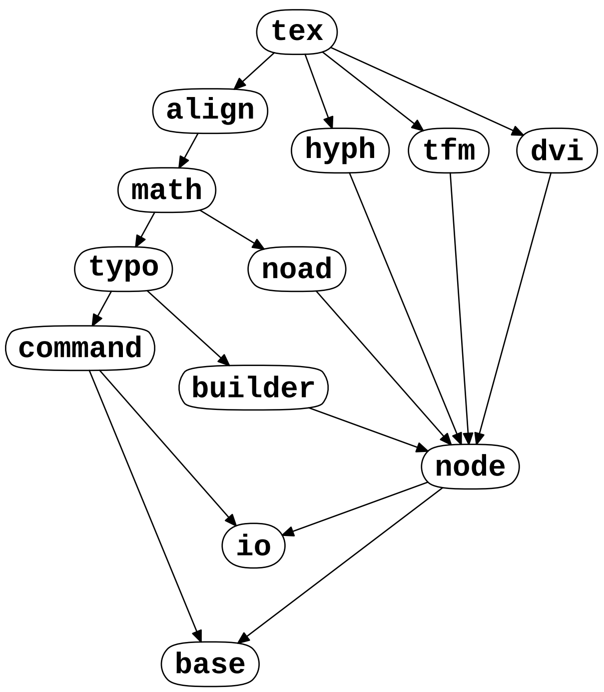

Figure 1: Dependencies of the packages in $\mathcal{NTS}$

share code between different types but also provides for the independence on the actual types from lower levels.

Compatibility *Compatibility is the ease of combining software elements with others.*[7]

Because $\mathcal{NTS}$ behaves like TEX and has the same inputs and outputs it should generally be compatible with the same applications which use TEX. TEX in turn is relatively easy to combine with other programs because it has a simple external interface. In batch mode it does not interact with the user at all hence only providing the inputs and recognising the outputs is necessary. TEX uses its own format for font metrics and typeset outputs (`.tfm` and `.dvi`) and $\mathcal{NTS}$ uses them as well. TFM can be easily converted by `afm2tfm` from PostScript font metric and DVI by `dvips` to PostScript.

It is possible to combine TEX with other software components at the programming level (like using `zlib` library from pdfTEX) but it is nearly impossible to use parts of the TEX code by other programs (reusability).

In contrast to TEX, portions of $\mathcal{NTS}$ can be used by other Java applications relatively easily. There are also many possibilities to utilise other Java packages by $\mathcal{NTS}$. Using existing packages for networking, compression, image manipulation or XML processing looks promising. These packages are a standard part of the Java platform and their existence was one of the important reasons for choosing Java.

Problems arise when combining Java code with code written in other languages. There exists an interface for calling native code (JNI) but unfortunately it is platform dependent. Calling Java code from native applications is even more complicated because the Java virtual machine is needed for interpreting the code.

Efficiency *Efficiency is the ability of a software system to place as few demands as possible on hardware resources, such as processor time, space occupied in internal and external memories, bandwidth used in communication devices.*[8]

The efficiency of current $\mathcal{NTS}$ implementation is a real disappointment. We used the processing of *The TEXbook* and reading and dumping of the ConTEXt format as benchmarks. In the former case $\mathcal{NTS}$ was approximately 30 times slower than TEX and in the latter the factor was even about 130. Just in time (JIT) compilation of Java bytecode was of course enabled and the jobs were reasonably long so the time overhead of internal compilation was amortised.

There are probably two reasons for this. One is the implementation language itself and second is the style of programming. In my opinion the true reason is the combination of both. The problem with the programming style is that it tries to be very clean and to express each design concept as a separate entity. The only means of doing it in Java is by defining and using new classes. This inevitably then touches a Java weakness which is the inefficient handling of many objects (class instances) and references.

One typical example of this problem is that even characters inside tokens and boxes are represented by special classes. My original idea was to use a basic Java type like `char` or `int` but this was rejected by the rest of the team as a type unsafe hack. Later in the project life I became to appreciate this decision because without such a separation the program could become a real mess.

I suppose that the source of the non-efficient object handling is that Java allocates all objects dynamically and keeps the references to them. No objects are statically allocated or embedded in other objects. This leads to costly allocations, garbage collections and over-linked representation of data structures even in cases when it is conceptually not necessary. It is my impression (albeit not deeply

[7] [7] p. 8

[8] [7] p. 9

founded) that the problem is inherent to the specification of both the Java language and Java virtual machine. An interesting discussion of the Java virtual machine and its suitability as a target for languages other than Java can be found in [2].

There are some possibilities to speed up $\mathcal{NTS}$. I suppose that there are not many places where more efficient algorithms can be used. In a good design it should be done in the right way mostly from the very beginning. Another possibility is to locally compensate the non-efficient Java object handling by caching objects like characters or character nodes which have many equal frequently created instances. Yet another possibility could be to replace certain classes (characters, dimensions) by primitive types. Unfortunately such hackery would certainly spoil the readability and type safety of the code and contradict the project goals.

The acceptable optimisations should be tried but I do not expect much from them. Java is known for its low efficiency which has been reported in connection with other applications. Even if we could increase the speed by the factor of five it would be still too slow to find a real user base. And what would the result be with $\mathcal{NTS}$ extented to solve more complicated problems?

Portability *Portability is the ease of transferring software products to various hardware and software environments.*[9]

The promised portability of Java was one of the main reasons for choosing it. There are a few problems when compiling of the $\mathcal{NTS}$ code with Java compilers from different vendors. These are caused by bugs in compilers or ambiguities in the Java language specification. The current $\mathcal{NTS}$ is a pure Java application and in principle we can say that Sun's promise: "Compile once, run everywhere" holds for $\mathcal{NTS}$.

Ease of use *Ease of use is the ease with which people of various backgrounds and qualifications can learn to use software products and apply them to solve problems. It also covers the ease of installation, operation and monitoring.*[10]

As far as typesetting of documents is concerned $\mathcal{NTS}$ is just as easy to use as the original TEX. Most probably, every reader does have a well established opinion about this level of easiness.

Well, the preceding paragraph was more of a joke. In fact the problem is much wider than just the processing of TEX documents. There are three important groups of potential users of $\mathcal{NTS}$. The first consists of implementors who might want to extend the program. Such extensions could be motivated by either pragmatic or theoretical and experimental reasons (or both). For those users the ease of use is a combination of (already discussed) extendibility of the system and the ease of use of the programming language.

The second group are the authors of macro-packages who would want to take the advantage of the extended functionality of $\mathcal{NTS}$ (to be discussed below). They will probably be interested in how easily $\mathcal{NTS}$ could support a better quality of their products. It will be of no surprise if the second and the first groups have a non-empty intersection.

Finally, the third group are the end users who might want to use $\mathcal{NTS}$ to typeset their documents. $\mathcal{NTS}$ is, similarly to TEX, not meant to be used directly. Most users use a macro-package like plain, LATEX or ConTEXt. For them the ease of use heavily depends on the ease of use of the respective macro-package.

The ease of installation should also be addressed here. $\mathcal{NTS}$ requires two other software packages to be run: the Java runtime environment (JRE) and a certain subset of the TEX directory structure (TDS). JRE is needed for interpreting of the Java byte code and from TDS it needs the font metrics (`.tfm` files), all TEX inputs for generating format files and the program `kpsewhich` to find these files. There is a good chance that at least TDS is already installed on the user's system. $\mathcal{NTS}$ itself can be distributed as a Java archive (`.jar` file) with compiled classes (`.class` files). This archive can be put somewhere into the filesystem with its location added to the environment variable `CLASSPATH`. If installing from source (`.java`) files you first have to compile the code using the Java development kit (JDK). All these steps are not very difficult.

Starting $\mathcal{NTS}$ is a little bit annoying because one has to type something like[11]:

```
java Nts -Dnts.fmt=plain mydoc
```

A script or batch file to wrap the verbose command would come very handy but it would be platform dependent. It is still open if the installation process can be automated lets say by clicking some link on the web and at the same time stay platform independent. Integrating $\mathcal{NTS}$ into existing TEX distributions should be quite easy provided that users have JRE already installed.

9 [7] p. 11
10 [7] p. 11

11 The format file `plain.nfmt` must first be generated by issuing: `java Nts 'plain \dump'`

Functionality *Functionality is the extent of possibilities provided by a system.*[12]

If the reader assumes that the $\mathcal{N_TS}$ functionality is the same as the well known TEX functionality and so there is not much to talk about, he is right. This is exactly the current state. Of course we want this to change in the future. The purpose of the reimplementation was to make it easier to build extensions providing new functionality.

It has already been mentioned that there are three groups of potential users. New functionality which can be directly visible to the end users will probably be the support of new, different, input and output formats. Allowing XML input, PDF output and support for more font metrics formats would be useful. Personally, I consider it to be more important to provide new primitives and modes for macro writers. This should become reflected in a new functionality on higher levels and/or in easier implementations and better efficiency and robustness of the existing stuff.

It may well be that some alternative input language could make typesetting easier or more general. And perhaps future will bring completely new ways of working with documents, different from the traditional TEX "programming" or WYSIWYG paradigms.

Timeliness *Timeliness is the ability of a software system to be released when or before its users want it.*[13]

It seems that the timeliness is my weak spot[14]. I have heard that some people had estimated that a prototype reimplementation of TEX in a high level fast prototyping language would take just a couple of months. $\mathcal{N_TS}$ is not a simplified prototype but a fully functional implementation. In fact it took a couple of months only to roughly analyse *TEX: The Program*, to sketch a basic architectural outline of the new design and to do the first trials of how the job could be done in Java. Moreover, Java is not a fast prototyping language and the prototyping approach has been sacrificed for the sake of platform independence and availability of useful standard libraries.

My contract stated that the project should be finished within one year. Other team members were aware of the fact that it was just an estimate because the whole task was an experiment. There was neither experience nor methodology for doing a reimplementation of this kind. After several reviews it became clear that the task is more complicated than expected. The main reason, in my opinion, was the requirement for $\mathcal{N_TS}$ functionality to be identical with TEX functionality. I have spent a great amount of time analysing tiny details of *TEX: The Program* code in order to find out what exactly would appear in the context trace of the next error message when some complicated macro is expanded or what would appear in `\showlists` output when some funny math construct is nested in a few boxes. And after that I have spent another great amount of time on testing of all the possible combinations of tokens and boxes, analysing the one-character differences in outputs, looking into *TEX: The Program* again and again and eventually fixing it. Conceptually interesting things like paragraph breaking or math formulas took surprisingly far less time.

Fortunately the first version is almost finished. It would be very interesting to know how much time it would take with the requirement of functional identity with TEX being not so strict and how useful the final product would be. It is very difficult to decide which TEX features do not need to be implemented faithfully as a lot of obscure cases which seem to be of no importance are actually used in "dirty tricks" even in major macro-packages. Therefore it is quite possible that more relaxed specifications could lead to faster coding but much longer testing and debugging. We will not know the answer unless somebody else tries it[15].

When looking at the exact wording of the timeliness definition it seems that an important aspect of the project has not been addressed: "Are out there any users who do really want $\mathcal{N_TS}$?" I think that the answer could be phrased as "Not really, not until it is able to provide something new." $\mathcal{N_TS}$ is not meant for quick and dirty solutions but for solving difficult problems which TEX cannot handle. There are some classes of problems which have already been well analysed (like weighted hyphenation) which can be implemented relatively quickly. Complicated problems like optimal page breaking or general multicolumn mode will probably need more conceptual work and can take years. Because TEX is still quite up to date and able to cover large areas of typesetting tasks the real $\mathcal{N_TS}$ is still a long term goal.

Discussion

Do the quality factors of $\mathcal{N_TS}$ meet our expectations? In my opinion it strongly depends on what

[12] [7] p. 12
[13] [7] p. 14
[14] also for this article
[15] There are also other projects which aim to reimplement TEX in less faithful way; I know about [1] and the TeXlib project initiated by Giuseppe Bilotta.

do we want from $\mathcal{NTS}$. But do we really know what to want? I have noticed that even the members of the $\mathcal{NTS}$ team had different ideas about the purpose of the project. There were many kinds of interests. Some wanted to make TeX more attractive and user friendly and they thought about providing a graphical user interface. Others hoped that $\mathcal{NTS}$ would eventually fulfill some concrete items from their typographer's wishlists. And yet others needed just a playground for academic experiments.

The quality factors studied above can have different priorities for specific purposes. If you want only to do experiments you do not care about efficiency (and in fact most of other factors) too much. If you want something user friendly then probably the ease of use is your first aim. The question is even more wide than just the quality factors. The first quality factor—correctness—is based on specification. But does the specification meet the purpose of the project?

Experience from the reimplementation shows that two points of the specification were problematic. They were mentioned several times in the evaluation of quality factors: the requirement for functional identity with TeX and the decision for Java as the implementation language.

Functional identity with TeX This requirement made the specification too rigid. It caused big difficulties in achieving correctness which in turn had a negative impact on timeliness. All this is already behind us and perhaps one could forget about the difficulties and appreciate the achievement. Unfortunately the factors which are important for future developments, mainly extendibility and reusability, could have been negatively influenced. Taking the good pieces from TeX is certainly an advantage but carrying the burden of bad bits makes the interfaces of $\mathcal{NTS}$ components too complicated. Is the strict compatibility with TeX really so important and worth the complications? There were several reasons for requiring the functional identity:

credibility The idea was that if we can prove that $\mathcal{NTS}$ is as good as TeX then the TeX users will trust it. But should this requirement have been so strictly followed? Would it not been enough to make $\mathcal{NTS}$ to be able to process all the TeX documents and to produce output of the same or even better quality? And why should the log output be the same by all means? What is wrong with the same information presented in a more readable way?

processing legacy documents Is this really necessary? This requirement has been formulated in times when the capacity of disks was limited and of course it would be essential if $\mathcal{NTS}$ had to completely replace TeX. Now that we have several TeX-like applications (ε-TeX, pdfTeX, Omega) in one distribution why not use TeX for legacy documents and $\mathcal{NTS}$ for new material? A complete replacement would be problematic anyway because $\mathcal{NTS}$ cannot work in a vacuum, it does need fonts and format files. Integration with the TeX directory structure is a natural solution.

convenient testing Testing is now really convenient because no differences are allowed. But would development of a few tools which could compare whether two DVI files produce the same visual output or whether two log files contain the same information be more difficult than the effort spent on achieving strict compatibility?

compatibility with macro-packages It is similar to credibility. The same semantics of input is important but the binary identity of the outputs is not.

It seems that compatibility with TeX is important but only to a certain extent. $\mathcal{NTS}$ should be able to process arbitrary documents because users have to use the existing macro-packages and to process existing documents. The DVI output should be visually the same now but in the future we can aim for a better one. It does not harm if the log output or reactions to errors are different. If applications exist which scan the log output (I am not aware of them) it should not be too difficult to rewrite them to scan log files of a different—possibly even more convenient—format.

Java When talking with people about Java and $\mathcal{NTS}$ implementation one notices that everybody complains that it is terribly slow. This would probably be acceptable if Java were outstanding in other aspects but this is not the case. The language misses some concepts and features which are important for real object oriented, modular and generic programming. It makes extendibility and reusability more difficult to achieve. Some of the reasons for choosing Java were:

great portability It is true if the application is in pure Java[16]. But then the possible interactions with the operating system are very limited. This leads to problems with configuration ($\mathcal{NTS}$ now calls `kpsewhich` when looking for files) and signal handling is impossible.

[16] without platform specific code in another language

easy distribution It is not clear if it is easy for users or for distributors. For users who do not have Java installed on their system, a decent distribution of native mode executables is more convenient.

appeal I am not sure whether there are more people who do like Java or those who hate it. It seems that within the TEX community the latter group is stronger.

high level It turned out that it is not as high a level as it appeared at the first sight.

Java community support Nothing happened.

standardisation With respect to some other languages, it probably is a strong point.

network awareness It is true that in Java it is probably easier to develop network aware applications or to build plug-ins.

It seems that Java has many drawbacks but this is true for all programming languages I know. Every language has also some merits. The only suggestion I have is that the choice of the language should be reconsidered very carefully against a set of well defined requirements.

The aims In my opinion catering for the various requirements is actually not so much of a problem. With $\mathcal{NTS}$ being really open many purposes could be served, even those whose existence we may even be unaware of. Problems would only arise with really contradicting requirements.

Although $\mathcal{NTS}$ should be as open as possible, priorities should be observed. One is the creating of useful and practical extensions which for example could be incorporating of the recent achievements present in ε-TEX, pdfTEX and Omega supplemented with some more already well understood features.

Another priority is long term and certainly most important for the future. $\mathcal{NTS}$ should provide a framework for experimentation and research in the field of typography. For that purpose a possibility of expressing the ideas in a language of a higher abstraction level would come very handy. Currently [1] is better suited for such purposes. If this aspect could be combined with practical functionality then outcome from research could be accommodated much easier.

My perception of $\mathcal{NTS}$ is not that of a single application. My idea which I did try to embody into the reimplementation is a library of software components. TEX-like $\mathcal{NTS}$ is just one particular application which uses the library. But even this application should be as open as possible. My dream is to provide in it a level of a high level language (like LISP) with general typographic primitives. That could be used either for convenient experimenting or for writing of such parts of macro-packages which need to manipulate typographic material and which now have to struggle with the macro-language. Such approach has already been proposed in [8].

Conclusion

$\mathcal{NTS}$ version one is almost finished. The good news is that it was designed to be highly extendible and the components to be reusable. Modulo few unimportant details it behaves like TEX. The bad news is that it is terribly slow and that the language of implementation and strict compatibility with TEX had decreased the openness and genericity of the code.

After the code becomes officially released anybody can take it and use it as a whole thing or just the components for his or her purposes. However, at the same time a canonical version should be maintained and development efforts should be coordinated. People interested in future developments or usage should clearly express their requirements in order to help make good decisions. The design, specification and implementation language all should be reconsidered in the light of the current and anticipated requirements.

I want to thank the TEX community for all support I have received and Donald E. Knuth for creating TEX to be complete in the sense that it perfectly matches *his* requirements.

References

[1] Blumensat, Achim. *ant V0.2*. Available at `http://www-mgi.informatik.rwth-aachen.de/~blume/Download.html`

[2] Gough, K. John – Corney, D. Evaluating the Java Virtual Machine as a target for Languages Other Than Java. In *Joint Modular Languages Conference, JMLC 2000*. Heidelberg : Springer-Verlag, 2000. Lecture Notes in Computer Science 1897, ISBN 3-540-67958-8, pp. 278–290.

[3] Hagen, Hans. Some $\mathcal{NTS}$ thoughts. In *EuroTEX'99 Proceedings*. Sep. 1999. ISSN 1438-9959. pp. 233–240.

[4] Knuth, Donald E. *The TEXbook*. Reading, Massachusetts : Addison-Wesley Publishing Company, 1986. ISBN 0-201-13447-0.

[5] Knuth, Donald E. *TEX: The Program*. Reading, Massachusetts : Addison-Wesley Publishing Company, 1986. ISBN 0-201-13437-3.

[6] Lammarsch, Joachim. The History of $\mathcal{NTS}$. In *EuroTEX'99 Proceedings*. Sep. 1999. ISSN 1438-9959. pp. 228–232.

[7] Meyer, Bertrand. *Object Oriented Software Construction, 2nd ed.* Upper Saddle River, NJ : Prentice Hall PTR, 1997. ISBN 0-13-629155-4.

[8] Mittelbach, Frank. E-TEX: Guidelines for Future TEX Extensions. *TUGboat*. Sep. 1990, vol. 11, no. 3, pp. 337–345.

[9] Skoupý, Karel. $\mathcal{NTS}$: a New Typesetting System. In *Proceedings of TUG'98*. August 1998. pp. 167–171. Also in *TUGboat*. Sep. 1998, vol. 19, no. 3, pp. 318–322.

[10] Skoupý, Karel – Taylor Philip. The implementation of $\mathcal{NTS}$. In *EuroTEX'99 Proceedings*. Sep. 1999. ISSN 1438-9959. pp. 246–260.

[11] Taylor, Philip – Zlatuška, Jiří– Skoupý Karel. The $\mathcal{NTS}$ project: from conception to implementation. In *Cahiers GUTenberg*. May 2000, no. 35–36, pp. 53–77.

[12] Zlatuška, Jiří. $\mathcal{NTS}$: Programming languages and paradigms. In *EuroTEX'99 Proceedings*. Sep. 1999. ISSN 1438-9959. pp. 241–245.