BibTeX++: Toward Higher-order BibTeXing

Laura Barrero Sastre

Fabien Dagnat
Computer Science Lab
ENST Bretagne, CS 83818
F-29238 PLOUZANÉ CEDEX
FRANCE
Fabien.Dagnat@enst-bretagne.fr
http://perso-info.enst-bretagne.fr/~fdagnat/index.php

Emmanuel Donin de Rosière
Emmanuel.DoninDeRosiere@enst-bretagne.fr

Ronan Keryell
Computer Science Lab
ENST Bretagne, CS 83818
F-29238 PLOUZANÉ CEDEX
FRANCE
rk@enstb.org
http://www.lit.enstb.org/~keryell

Nicolas Torneri
Nicolas.Torneri@enst-bretagne.fr

Abstract

In the LaTeX world, bibTeX is a very widely used tool to deal with bibliography notices. Unfortunately, this tool has not evolved for the last 10 years and even if few other bibliographical tools exist, it seems interesting to develop a new tool with new features using modern programming standards.

The bibTeX++ project began in 1999 and is written in Java for the portability and the object oriented aspect and offers new functionalities:

- for the expressiveness, the security model and native unicode character set support, bibTeX++ styles are directly a Java class;
- for compatibility, it uses advanced compiler techniques to translate plain old bibTeX styles to new Java styles. Such a translated style can even set the development basis of a new native bibTeX++ style to ease the style programmer's life;
- the architecture is designed for extensibility and split in different components: core, parsers (to be compatible with other bibliographical styles, sources or encoding schemes), pretty printers (to generate bibliographies for other tools than LaTeX), plugins;
- the plugin concept is used to dynamically extends bibTeX++ functionalities. For example, since (meta-)plugins can load new plugins, new styles can be directly downloaded from the Internet.

– – * – –

Introduction

A bibliography or list of references is a listing of all sources from which you have taken information directly (by literal quotation) or indirectly (through paraphrase), or where you have used information or reproduced material from. These references are used to:

- clearly identifying each document. So it provides some identification elements to allow the reader to look for these documents in library catalogues or anywhere else. In most cases, these identification elements are normalized (ISO, 1958) (we often use the name of the book,

the author, the editor,...), but because of the numerical revolution, there are more and more document types (web pages for example) and identification elements (e-mail, URL,...). So the management of the references became more and more difficult;

- enable the reader to consult the sources you have used with a minimum of effort. Thus we have to indicate precisely where (on which page, the electronic location) or under which circumstances (personal interview, e-mail) you obtained the information.

Unfortunately, creating a bibliography has always been an headache for typographers: if the article talks about "(Larousse, 2002)" as a book introducing meta-middleware and in the bibliography "(Larousse, 2002)" is described as a French cook book instead, there has been clearly a problem somewhere. Another problem is that each journal has its own bibliography style, so a bibliographical management software has to allow the user to create his own style and send it to other people.

Fortunately, BibTeX (Patashnik, 1988) is such a popular tool used by the LaTeX (latex, 2004) community to generate bibliographical notices in publications. If there are also many other tools available (Donin de Rosière, 2003b), this one suites very well the LaTeX philosophy and is well integrated to it: the user describes what she wants with a mark-up language without having to dive into the deep layout details: the LaTeX infrastructure will use some styles to typeset the presentation from the high level content description.

The citation matter is stored in a database (a file with a `.bib` extension) and BibTeX picks from the `.aux` file the needed information according to the citation marks placed in the LaTeX document (`.tex` file) by the author. A typeset version of the bibliography is done by BibTeX to the `.bbl` file by using a `.bst` bibliography style file. The work-flow is summed up on figure 1 (Goossens, Mittelbach, and Samarin, 1994).

There are a lot of bibliographical styles available for BibTeX targeted at many different scientific journals, book styles, etc. and even one for *TUGboat*which is used here. The user needs only to select the desired style and the bibliography notice is generated from her common bibliographical reference database. There are also a lot of such bibliographical reference databases available on the Internet that can be used directly, such as (Lawrence, Giles, and Bollacker, 2002).

There are many tools targeted at easing the management of all this BibTeX files: database tools,

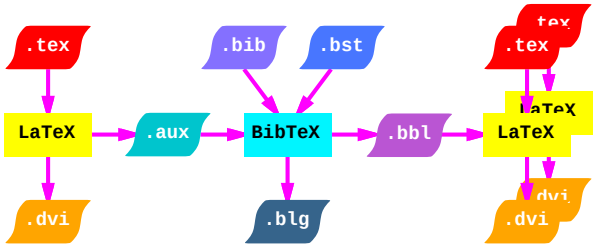

Figure 1: BibTeX work flow.

editors,... For example to write this article we used the grand Emacs multi-platform text editor that has various modes to deal with a LaTeX document in various coding systems: the great AUCTeX mode (Abrahamsen and Kastrup, 2004) to deal with many things in the LaTeX source file, the refTeX (Dominik and Eglen, 2004) to deal with citations and cross-referencing such as a bibliography browse-and-pick mode and some BibTeX editing modes.

A lot of BibTeX record databases are also widely available on the Internet through bibliographical servers such as CiteSeer (Lawrence, Giles, and Bollacker, 2002) and it is often easy to get some BibTeX records from few keywords.

Even there are so much material available for BibTeX, BibTeX is old, does not evolve anymore and has many shortcomings according to modern tool standards: the character encoding is constrained to 8-bit ASCII variant, it is difficult to mix different languages on the database side or on the document side, memory usage is selected at compile time, style programming is done in a language that is rather optimized for the implementation efficiency and not for the style programmer brain. The tool lacks extensibility and modern features as network awareness.

It is why we began the BibTeX++ project in 1999 as a way to extend the BibTeX functionalities without throwing away the BibTeX compatibility, databases and styles.

In this article we present first the basic functionalities in BibTeX++ (§ 'BibTeX++ functionalities'), then the software architecture in § 'Software architecture of BibTeX++' with some emphasis on the BiSTro compiler ('Software architecture of the BiSTro BST to Java compiler') and the plugin architecture ('Plugins and meta-plugins').

BibTeX++ functionalities

The BibTeX++ name has been chosen to assert a BibTeX compatibility with an improvement comparable to the object oriented add-on from the C to C++ languages. If the plain BibTeX++ user

does not embrace the object oriented architecture of BibTeX++ it is not the case of the style programmer (and of course the BibTeX++ programmers).

BibTeX compatibility Since BibTeX is a very widely used tool, the compatibility with the old BibTeX is a requirement.

BibTeX usage is simple and the first thing to do is to create the database file (a file with a `.bib` extension) or better to use an old one with at least the bibliographical references that are used in the article to produce with LaTeX. Each reference record has a type chosen from the 13 possible default types (article, book, conference, thesis,...) and is composed from various fields such as its key (used to cite this reference from the LaTeX file), its title, its author(s), the publication year and so on such as:

```
@BOOK{LaTeX:companion,
      Author = {Michel Goossens
                and Frank Mittelbach
                and Alexander Samarin},
      Publisher = {Addison-Weslay},
      Title = {The \LaTeX{} Companion},
      Year = 1994
}
```

This reference describes a book and can be cited with the key `LaTeX:companion`. If there are several authors, they are separated by an `and`. Plain LaTeX can often be put in most of the fields such as the `\LaTeX{}` in the previous `Title`. There are numerous kinds of fields (23) to be used to define a reference. According to the reference type, a field can be mandatory or optional. Other fields can be used for extension or information since they will be ignored by BibTeX itself.

Once the reference file is created, the reference must be inserted in the LaTeX document. For this purpose some `\cite{`⟨*key*⟩`}` are put where a reference to ⟨*key*⟩ is to be cited. Thus with the previous example inserting in the text "`\cite{LaTeX:companion}`" will appear as "(Goossens, Mittelbach, and Samarin, 1994)".

One instructs LaTeX to use a bibliography ⟨*style*⟩ with `\bibliographystyle{`⟨*style*⟩`}` and to insert the bibliography somewhere in the document from the database ⟨*bib-base*⟩`.bib` with a `\bibliography{`⟨*bib-base*⟩`}`.

When run, BibTeX picks in the ⟨*document*⟩`.aux` file the needed information given by the citation marks placed in the LaTeX document (⟨*document*⟩`.tex` file) by the author. A ready-to-typeset version of the bibliography is done by BibTeX to the ⟨*document*⟩`.bbl` file by using a ⟨*style*⟩`.bst` bibliography style file and the citation

```
FUNCTION {sort.format.names}
{ 's :=
  #1 'nameptr :=
  ""
  s num.names$ 'numnames :=
  numnames 'namesleft :=
    { namesleft #0 > }
    { nameptr #1 >
        { "   " * }
        'skip$
      if$
      s
      nameptr
      "{vv{ } }{ll{ }}{  ff{ }}{  jj{ }}"
      format.name$ 't :=
      nameptr numnames = t "others" = and
        { "et al" * }
        { t sortify * }
      if$
      nameptr #1 + 'nameptr :=
      namesleft #1 - 'namesleft :=
    }
  while$
}
```

Figure 2: BST sample code.

matter found in the `.bib` file(s). Error and information output goes to the ⟨*document*⟩`.blg` file. The work-flow is summed up on figure 1 (Goossens, Mittelbach, and Samarin, 1994).

BibTeX comes with some predefined styles but it is also possible to program other styles, for example to accept an `url` field to cite Internet stuff. All these styles are defined in some `.bst` files that are used by BibTeX to generate the bibliography according to the work flow described on figure 1.

From the user point of view, these concepts are also used in BibTeX++ *texto* with all the old BibTeX styles available too.

The style files are written in the BST language[1]. This is indeed an awful stack based language rooted in the sixties that has been chosen for an easy implementation of BibTeX concepts: it is simple to parse and to execute on a computer. The dark side is that the programming burden is put afterwards on the style programmer. A BST code looks like the figure 2 and there are 1258 such lines in the classical `alpha.bst`...

It is clear that building a new style from scratch is a *tour de force* and often basic programmers will

[1] According to its author, this language has no name in fact. But for the clearness of this article we call it "BST".

change only few details. Fortunately there are many styles available already than can fit most of the needs and there are also some custom styles generators (Daly, 2004).

Extensions to BibTeX A basic extension is to be able to deal with another encoding than the plain ASCII and more generally to be multilingual. Since BibTeX can sort the bibliography, a localized sort according to the document language is to be introduced.

A tool designed today could hardly escape the network option, so BibTeX++ must have a way to access world-wide on-line bibliographical database through the Internet.

The BST language in BibTeX lacks some expressiveness and a new language should be designed instead of trying to extend the old one. Some modern features should be added to allow scalability and extensions, for example by using an object oriented language. But we also want a compatibility with the old BibTeX styles that are written in BST, so this is rather tricky.

Besides BibTeX targets only LaTeX, BibTeX++ could also target other typesetting tools or other bibliographical database concepts.

But since BibTeX++ is still a work in progress as a general bibliographical workbench, there are many new functionalities that are not yet implemented or even not envisioned yet.

Style programming in BibTeX++ If we have to define a new language for BibTeX++ it should be a clearer language than BST.

A good candidate is to choose a domain specific language (DSL). For a programmer point of view it is yet another (less) cryptic language to learn that is still cumbersome. The expressiveness may not suite everyone needs and we may extend the language later to fit some usages.

On the other side, if we want a language as expressive as any another computer language, why not using such a language? If we choose an object oriented language, all the domain specific aspects could be seamlessly hidden in objects dealing with all the bibliographical stuff.

Since in BibTeX++ there is no particular performance requirement, this object solution is acceptable.

The next question is to select a implementation language. We want BibTeX++ to be portable, programmed in a clean language from a syntax and object point of view that can deal with big programs. The language should come with a lot of standard library to deal with all the modern programming ways (UNICODE, Internet,...), spread enough to avoid the *yet another weird-language to learn* syndrome.

Of course there is no one-stop answer and we cannot escape some trade-offs. From our point of view, Java has been considered as a good candidate.

The BibTeX++ core is thus directly written in Java for expressiveness and simplicity. All the generic library functions and classes to deal with bibliographies in BibTeX++ have been rewritten in Java too. BibTeX++ can run on every machine for which we have a run-time and a compiler.

For the internationalization, UNICODE is natively accepted in Java, we inherit all the Java locale stuff and even specialized collators for international sorting so important in BibTeX.

Of course choosing a different language from the original BST language does not solve at all the BibTeX compatibility issue. This one can be solved by implementing BibTeX as an add-on in BibTeX++, which remove a lot of interest in BibTeX++ or add a translation process from BST to the new programming style in BibTeX++.

This last approach is more challenging but far more interesting since the translated old BST styles can be used as the basis of new style developments. This translation process is deeper described in § 'Software architecture of the BiSTrO BST to Java compiler'.

Extending BibTeX++ further: plug-ins and meta-plug-ins Extensions in the old BibTeX to deal with new concepts are quite difficult to develop since code must be added directly in the BibTeX source.

In a modern tool, it is, mandatory to use a more incremental and tractable approach even for a non experimented user by allowing loadable add-ons or plugins instead of needing to dig into the code to change its behavior. A plugin is a piece of code (a module) that can be added to BibTeX++ to increase its functionality without having to change the native BibTeX++ infrastructure.

Plugins should be able to modify any BibTeX++ behavior without puzzling the original design. Of course there is compromise to be found between expressiveness and complexity.

We present further the kind of extension one can expect with the plugin concept applied to different parts of BibTeX++ (described later in § 'Overview of the BibTeX++ architecture').

Parsers A natural extension need is about the input syntax that BibTeX++ can accept. New

parser can be written and used to change any information source into some internal abstract representations of the tool.

The bibliographical source could be changed to directly use the CiteSeer database (Lawrence, Giles, and Bollacker, 2002) through Internet, use some XML database or any other bibliographical database tool instead or in addition to the old `.bib` syntax database.

The input selection, normally read from the `.aux` file to pick `\cite` information could be extended to deal with OpenOffice, DocBook or Word™ documents.

Prettyprinters Since BibTeX++ can be seen as a parameterized compiler that deals with some internal bibliographical representations, it could be nice to target other output formats than LaTeX such as an OpenOffice, DocBook or Word™ document syntax or to automatically internationalize a style from one language to another one.

In an easier way BibTeX++ could be used to translate a `.bib` output to another bibliography database format or apply some basic manipulations on bibliographical databases (sorting, merging,...).

Style transformation More generally, adding some features to already existing styles is useful to revival some old-fashioned style, such as dealing new `url` records to old styles written before the Internet wave.

To make bibliographical database or a researcher publication list (Keryell, 2004) available on the Internet it could be useful to be able to fetch close to the typeset bibliographical notice the `.bib` record itself that was used to generate the notice if someone wants to cite it in her article. This could be done by using with any style an appropriate plugin.

Metaplugins Whereas BibTeX did not evolve for many years, BibTeX++ should rapidly evolve according present programming and Internet standards. A natural way is to use mobile code concepts in BibTeX++ that can also be downloaded by a plugin itself. Since it is a plugin than can fetch from the network other plugins it has been nicknamed meta-plugin in BibTeX++.

One can embrace various future use of this concept.

First, to write this paper we could have picked all the BibTeX++ bibliographical style from the *TUGboat* web site.

On the BibTeX++ web site we could have a page to reference all the bibliographical databases available on the Internet. A plugin associated on this page could download other plugins to deal with each database we are interested in.

If a new typesetting tool is introduced, its author could provide on her site some plugin matter for BibTeX++ to be compatible with that new tool. The BibTeX++ user would only provide to the meta-plugin the plugin address.

We will see in § 'Security model' how to circumvent the obvious security issues related with this kind of mobile code in the wild.

Plugin syntax To remain close to the actual LaTeX and BibTeX interaction syntax we do not add new LaTeX macros but rely on existing one just by adding special keywords inside `\bibliography` or `\bibliographystyle` macros.

Of course plugins can also be included from the BibTeX++ invocation line or with another mechanism to suite other input formats than LaTeX.

To avoid conflicts with some other tools that could also use this kind of extensions, a naming space beginning with `:bibtexpp` is used.

For example if a user from the computer science community want to directly used citations from the CiteSeer database (Lawrence, Giles, and Bollacker, 2002), this will be chosen with

```
\bibliography{:bibtexpp:plugin:citeseer}
```

If a user want to add a plugin style to automatically add the output of an `url` attribute if any from the database at the end of the typeset bibliography item, the following will have to be added

```
\bibliographystyle{:bibtexpp:plugin:
  add-attribute:url}
```

or more generally to add the output of some specific attributes `description` and `summary`

```
\bibliographystyle{:bibtexpp:plugin:
  add-attribute:list:description,summary}
```

or all the attribute with

```
\bibliographystyle{:bibtexpp:plugin:
  add-attribute:all}
```

Plugins can be downloaded from the default BibTeX++ network repository defined in its code with for example

```
\bibliographystyle{:bibtexpp:plugin:meta:
  beautiful-bib}
```

of from any other place by defining an URL such as

```
\bibliographystyle{:bibtexpp:plugin:meta:
  URI:http://nice.bib.org/beautiful-bib}
```

If the BibTeX++ installation is a little bit old, some plugins may be absent from the local distribution but present from the BibTeX++ Internet repository. To be able to compile documents without choking BibTeX++ can be used in an automatic metaplugin way where any lacking plugin will be retrieved from the repository.

If other plugins have registered on the naming global BibTeX++ directory but reside on other servers, a proxy-plugin will be downloaded to download later the real plugin code from its server.

Naming clash should be avoided either with the current LaTeX best practice for package names or by using a hierarchical naming space mimicking Java class naming space or URL names.

Indeed a plugin call can be defined in either the `\bibliography` or `\bibliographystyle` macros in an indifferent way but, according to the plugin role, one may be more logical than the other.

The basic compatibility with LaTeX and BibTeX is based on the fact that some macros with this syntax will only generate warning in BibTeX without **refraining** the LaTeX compilation. Of course, the bibliography of the document will be lacking or at least incorrect, a user without BibTeX++ will be able to have an approximation of the document.

More expressiveness can used with new macros defined in a new package `bibtexpp` if the previous relative compatibility is not mandatory.

Of course, a parser plugin can itself define a new plugin definition or parameter syntax to deal with other input languages such as with DocBook or Word™ documents.

Software architecture of BibTeX++

Overview of the BibTeX++ architecture In BibTeX++ we are specially interested in the software architecture point of view since the project began first with the challenging idea of compiling BST language in something newer and more expressive.

The principal elements of this architecture are closely related to the BibTeX++ data-flow and is summed-up on figure 3.

To be more generic and more scalable, output are dealt with pretty-printers of internal representations and the inputs are read by parsers that build a common internal representation. In this way, we have only to redefined a new pretty-printer or parser to deal with a new format without having to change the BibTeX++ internals.

The parsers are made with SableCC (Gagnon, 1998), a compiler compiler for Java written in Java. Unless other compiler compilers (like Java Cup for example citecup:manual), we do not need a specific library to use the generated compiler. So users only need Java to execute BibTeX++.

- The AUX parser which tries to obtain the name of the style, databases filenames, and information about languages used in the LaTeX file.
- The BIB parser which converts databases to a list of Java objects.
- The BST parser which transforms style file into a syntax tree.
- The BST compiler. It takes this syntax tree and makes a Java style file from it. It also tries to correct some usal errors in BST files and optimize output code.

BibTeX++ core The core deals with all the basic BibTeX++ infrastructure from bibliographical concepts to house-keeping and the data structures used by various abstract internal representations.

All the BibTeX functionalities that can be used by the various bibliographical style are implemented in a core library.

Bibliography back-end This easier part output the internal style execution into a file usable by the typesetting tool to be used. Right now, a `.bbl` file to be used by LaTeX is generated but by changing this pretty-printer other output could be generated for an other tool.

Document front-end It is organized according to a strategy pattern to be able to deal with various document formats.

Right now only a SableCC parser has been written to deal with LaTeX.

Bibliographical database front-end This front-end follow also a strategy pattern to cope with various database format.

But only a SableCC parser has been written now to read BibTeX bibliographical database in the `.bib` format.

Caching BibTeX++ styles BibTeX++ style can be stored on the computer running it as for BibTeX but can also retrieved from the network or translated from an old BibTeX BST style.

If the first access is quite fast, the two other ways may be deadly slow compared to it. It is why a cache architecture has been added in the BibTeX++ style pipe to avoid fetching again and again a remote BibTeX++ style or translating a style from BibTeX format to BibTeX++ at every BibTeX++ run.

Software architecture of the BiSTro BST to Java compiler In BibTeX++ we are specially interested in the software architecture point of view since indeed the project began first with the challenging idea of compiling the BST language in something newer and more expressive.

The BibTeX stack-based input language BibTeX++ `.bst` files use a stack-based language: it is a type of language where you have to put the data you want to use on a stack. Functions store also their results on this stack. The syntax uses an inversed polish notation (or postfix notation). For example, if you want to do compute `2 + 3`, you have

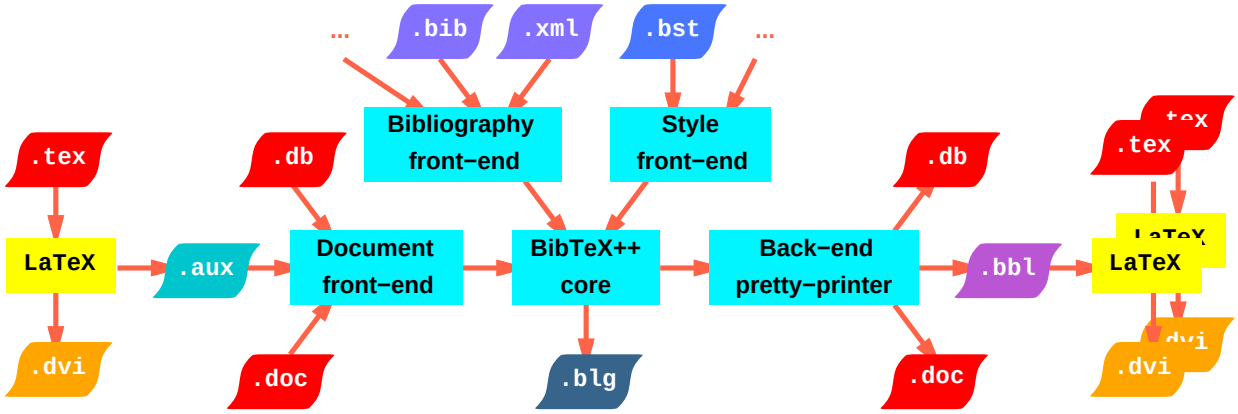

Figure 3: BibTeX++ work-flow.

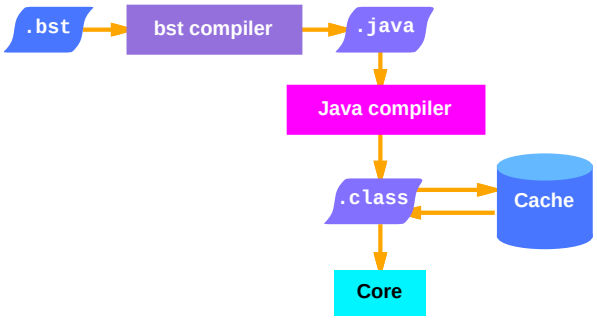

Figure 4: Compilation work-flow.

to put 2 and 3 on the stack and then call the `add` operator, so you have to write something like: `2 3 +`.

Some stack-based languages We may cite as stack-based programming languages:

- Forth used in many fields, especially in embedded control applications;
- PostScript used primarily in typesetting and other display purposes. But because the majority of PostScript code is written by programs, it can also be regarded as an intermediate language;
- RPL (reverse polish LISP), the language of the HP-48 calculator. It is a run-time type-checked language with mathematical data types (it is on a calculator) and has a Forth-like syntax;
- the BibTeX style language or BST language, the stack-based language used for processing BibTeX databases.

But even stack-based languages are sometimes used as programming languages, they are more popular as intermediate languages for compiler and as machine-independent executable program representations. As intermediate languages may be listed:

- the UCSD P-code used in the UCSD system, an operating system well-known for its Pascal compiler. P-code was either interpreted or compiled to native code. We can find today P-code compilers for more recent system;
- the Smalltalk-80 byte code is the intermediate language of the Smalltalk-80 system.

Some other stack-based languages target at several machines as machine-independent languages such as:

- the JAVA bytecode output after compiling a JAVA file. It can be used on every system supported by JAVA because it will be interpreted by the JVM (JAVA Virtual Machine).

Unfortunately computers are mainly register machines[2] and it is not easy to implement directly and efficiently a stack-based language. That is another reason why stack-based language are not frequently used. Today, there are 3 ways to execute a stack-based language on a register machines by using:

- an interpreter. The execution is dynamic but very slow;
- a compiler that statically transforms the code into target one;
- a source-to-source translator to convert stack-based code into a high-level language that is then compiled for the target. It allows the code to be executed on many different systems if the high-level language is well-known and widely used.

[2] There is probably no popular architecture since the Transputer (Inmos, 1989).

BST language A style file for BibTEX is a program that formats the reference list in a certain way. For example, a style file can sort the reference list by alphabetical order thanks to the autor names and makes the title in italics.

The BST language (Patashnik, 1988) uses ten commands to manipulate language objects (constants, variables, functions, the stack and the entry reference list). A string value is between double-quotes like `"abcd efgh"` and an integer is preceded by an `#` like `#23`. There are also three different types of variables:

- global variables (declared by an `INTEGERS` or `STRINGS` command);
- entry variables which can be strings or integers, with a value assigned for each entry of the list;
- fields which are read-only strings. They represent information from the curent reference item, so each one has a value for every entry.

Since BST is a domain-specific language there are among the 10 commands available some more interesting ones:

- `ENTRY` which declares the fields (in the bibliography databases) and the entry variables. `crossref` is a field which is automatically declared (used for cross referencing) and `sort.key$` is an entry variable (used for sorting references) also automatically declared;
- `ITERATE` which executes a single function for each entry in the reference list. These calls are made in the list current order;
- `READ` which reads the database file and assigns to fields their value for each entry;
- `REVERSE` which performs the same action than `ITERATE` but in reverse order;
- `SORT` which sorts the reference list in alphabetical order according to `sort.key$`.

All these commands permit to define the structure of a style file. But with them, we can not manipulate variables. It is why 37 built-in functions have been declared in *BibTEX*, from integer and string computations to control-flow operation (`if$`, `while$`,...) and the `write$` that writes the top string item into the `.bbl` file (the BibTEX output). With all these built-in functions and commands some other new useful and more complex functions can be designed such as:

```
FUNCTION {and}
{   'skip$
    { pop$ #0 }
  if$
}
```

This function calculates the "logical and" between two numbers: if the first element on the stack is greater than 0 (that means "true"), `skip$` is executed so this function returns the second element on the stack. Else, `pop$ #0` is executed which puts 0 on the stack. We can see that, even with this oversimplistic example, it is not quite easy to understand the BST language:

- we are not accustomed with this postfix stack notation;
- the number and the type of input and output variables are implicit;
- we have to read all the control structure in reverse order.

It is the reason why only few people are able to program a new style in this language. So for BibTEX++, we will have to create a style language more expressive. But, because of the need of compatiblity with *BibTEX*, we'll have to transform this stack-based language into a standard one. So we will see how to remove the stack in a stack-based language.

Stack removing in stack-based language According to the translation type, several techniques has been proposed in the literature.

Source-to-source translator There are only few source to source translators for stack-based language. The most famous research on this was made in (Ertl, 1992; Ertl, 1996) where Forth code was translated into C language in order to increase the portability of Forth applications. Actually, to use a Forth application on a special system, one should develop a special interpreter. However, if one transforms the Forth into C first, the program can be used on every system where a C compiler is available. Furthermore, no deep optimization of the translator is needed since the C compiler will optimize the output code.

Practically all the other source to source translators for stack-based language are JAVA decompilers like *krakatoa* (Proebsting and Watterson, 1997) or *mocha* (Vliet, 2003) which try to transform JAVA bytecode back into JAVA. With this, the idea is to get back the program sources from compiled files. Nevertheless, all these translators use the same algorithm and special optimizations for JAVA and JVM (JAVA Virtual Machine) bytecode.

In (Ertl, 1996) the *f2c* Forth to C translator uses several steps to convert Forth code to C language. The first one is to split the code into its functions which will be processed independently. Then, *f2c* counts for each function the number of input and output parameters. The next step is to convert the

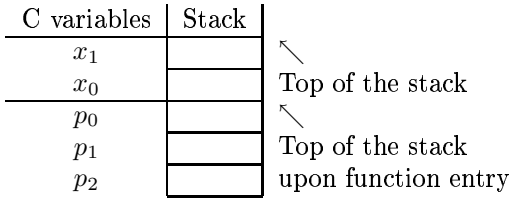

Figure 5: Stack mapping to C variables.

elements on the stack into C's local variables with the notation of the figure 5.

So, p_0, p_1,... represent the entry variables of each function and x_0, x_1,... are used like local variables. This scheme ensures that stack items that are not affected by an operation do not have to be copied around between local variables.

Then *f2c* converts each Forth primitive into a C sequence. For exemple, if the top of the stack resides in x_1, the translation of `+` will look like:

```
{
  Cell n1=x1;
  Cell n2=x0;
  Cell n;
  n = n1+n2;
  x0=n;
}
/* top of the stack now: x0 */
```

This sequence is very long, but a good C compiler can compile it to only one instruction (sometimes, it can convert several sequences into one instruction). So the translation process always works like this:

- all the usefull elements are declared as C local variables and are initialized;
- the C code for the Forth primitive is generated;
- the result variables are copied back to the stack.

f2c has also to convert all the control structures. But Forth allows the creation of arbitrary control structures so it is easier to convert them into `goto` C instructions and labels.

Nevertheless, this translation mechanism needs to know the height of the stack everywhere in the Forth code, but it is not always possible. Sometimes the stack depth is unknown. For exemple, in the case of an instruction like `?DUP 0= IF` which means that if the top of the stack is `0`, we remplace it with the previous element on the stack, else we delete the element. So in this case, *f2c* has to create a C stack and uses it in the whole function.

Compiler Source-to-source translators are not the only softwares which remove the stack in a stack-based language. Stack-based compilers do the same thing, but they convert this language into a low level one (often into mnemonic instructions). So their algorithms may be useful for BibTeX++.

RAFTS (Ertl, 1992) is a framework for compiling Forth code. It tries to produce fast and efficient code, so it needs to use some optimization techniques and interprocedural register allocation to eliminate nearly all stack accesses because they slow down the execution of the program. RAFTS compiles all of Forth, including unknown stack heights.

RAFTS uses several steps to compile Forth code. The first step is to split the code into basic blocks. A basic block is a set of instructions which contains only simple primitives like literals, constants, variables, operators and stack manipulation instructions. So a basic block does not contain any branch or jump: all primitives are executed sequentially. Then RAFTS builds a data flow graph of this basic block.

Just after that, it converts Forth primitives into mnemonic instructions and transforms all stack items into unlimited pseudo-registers. So all stack accesses within a basic block have been eliminated and the DAG (Directed Acyclic Graph) is now an instruction DAG. Then an instruction scheduler orders the nodes of the instruction DAG, i.e. it transforms the DAG into a list. This list is optimized for reducing register dependencies between instructions.

Now, we have a set of mnemonic blocks, but we have to connect them thanks to control structures. Control flow splits (`IF`, `WHILE` and `UNTIL`) are easy to transform but control flow joins (`ENDIF` and `BEGIN`) are a little harder because the corresponding stack items of the joining basic blocks usually do not reside in the same register. So RAFTS needs to move some values around to have the same structure.

In order to have a faster output code, three good register allocation algorithms are proposed: graph coloring register allocation (Briggs, 1992), hierarchical graph coloring (Callahan and Koblenz, 1991) and interprocedural allocators (Chow, 1988).

An other stack elimination in a compiler can be found in JAVA compilers. Today, faster and faster execution is needed for JAVA applications. Higher JAVA performances can be achieved by *Just-In-Time* (JIT) compilers which translate the stack-based JVM bytecode into register-based machine code. One crucial problem in JAVA JIT compilation is how to map and allocate stack entries and local variables into registers efficiently and quickly as to improve the JAVA performance.

LaTTe (Yang, Moon, and Altman, 1999) is a JAVA JIT compiler that performs fast and efficient register mapping and allocation for SPARC machines.

LaTTe converts Java bytecode (a stack-based language) to sparc mnemonic. It uses severals steps for this:

- first, LaTTe identifies all control join points and subroutines in the Java bytecode via a depth-first traversal in order to build a control flow graph (cfg);
- then, it converts this bytecode into a cfg of pseudo sparc instructions with symbolic registers;
- Optionally, some traditional optimizations are performed;
- in the fourth step, LaTTe performs a fast register allocation, generating a cfg of real sparc instructions.
- finally, the graph is converted into a list of sparc instructions.

In order to transform the stack into registers, LaTTe uses symbolic pseudo-sparc registers whose names are composed of three parts:

- the first character indicates the type: `a` for an address (or object reference), `i` for an integer, `f` for a float, `l` for a long and `d` for a double;
- the second character indicates the location: `s` for operand stack, `l` for local variable and `t` for temporary variable used by LaTTe;
- the remaining number distinguishes the symbolic registers.

For example, `il2` represents the second local integer register. But at the end of the algorithm, LaTTe transforms these pseudo-registers into real ones with two passes for each extended basic block:

- the backward sweep algorithm is a post-order traversal which collects information on the preferred destination registers for instructions;
- the forward sweep algorithm is a depth-first traversal which performs the real register alloation using that information.

Sometimes, we need to move some registers in order to reconcile register allocation at region join points because LaTTe use these two algorithms on each extended basic block independently. So two blocks may not use the same register for the same item on the stack.

Globally, this method is very efficient: the output code of LaTTe is on average two times faster than with the Sun jit and this speed comes particularly from the register allocation algorithm.

Compiling BibTeX bst style to BibTeX++ Java styles The transformation of a typeless stack-based language into an object-based one is something quite unusual and a bit complex but we planed a classical compiler architecture divided in several steps as shown on the figure 6:

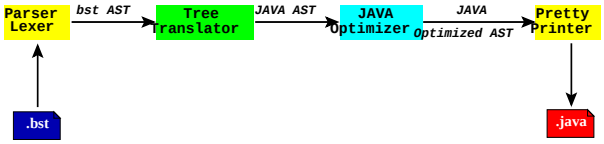

Figure 6: BiSTrO architecture.

- at first, we use the bst parser to convert bst file into a bst language abstract syntactic tree (ast) with a SableCC grammar;.
- then in the tree translator the bst ast is transformed in a Java-like ast :
  - useful information from this tree like function names, global variables,... are gathered. Functions are analyzed to decide if the stack can be removed and if so what are the number of input-output parameters;
  - unlike bst style, Java is a type-based language, so we need to know the type of every variable in non-stack-based session. But in some cases, it is very difficult to find this type, so if we cannot know it, we use instead a *cell* object: an object that can store both a integer and a string.
  - With those data, the bst ast is translated into a Java ast ;
- later the Java ast tree is optimized with some classical transformations such as constant propagation, dependencies reduction, transformation of integer into boolean in the condition block, dead code elimination, peephole optimizations,...
- At last, Java file is wrote by the pretty-printer and the new Java BibTeX++ style can be compiled and used.

Globally, some bst code like

```
FUNCTION {or}
{   { pop$ #1 }
    'skip$
  if$
}
```

will be converted into Java as

```
public int or( int i0 , int i1 )
{
   if( i1 > 0 )
   {
      i0 = 1;
```

```
    }
    return( i0 );
}
```

For most users and style designers, since the second code has been optimized for human comprehension, it should look more comprehensible: it will be easier to modify an existing style as a development basis of a new native BibTeX++ style.

Furthermore, because there are more Java programmers than BST ones, new arbitrary and complex styles will be easier to create with BibTeX++ than with BibTeX. If a simple interpretor had been designed instead of a translator, this could not have been possible.

More information on the BST to Java compilation can be found in (Donin de Rosière, 2003a) but 2 phases are detailed here.

.

Java translation First information about functions are searched through the BST AST. For each function, we try to obtain the name of the function, the number of input and output parameters and the possibility of removing the stack in this function: unfortunately this is not always possible.

Then the type of the arguments of all functions are inferred with type propagation from hints found in the program outside the stack, such as typed constants (a string or an integer), typed global variables, use of BibTeX functions with well-known entry or return types. The propagation is recursively done for all the home-made functions. Propagation is done in both direction in a use-def or def-use way to deal with typed entries or output. Some further abstractions are used to follow the dependence graph even if there are stack manipulation operation in the code such as `duplicate$`, `pop$` or `swap$`.

When it is not possible to infer the type, it indicates that we will have to use a polymorphic *Cell* object which can store either a string or an integer.

Next the body of the functions and their stacks are analyzed to find how many Java local variables we will have to use and what are their types.

With all this information, the BST code is translated to Java functions where we can remove the stack by using local variables instead of stack items if it.is possible or generate Java functions with a Java stack when stack removal is not possible. Variables are named accordingly to the figure 7. If the type has been inferred the native Java types `String` or `int` are used instead of our polymorphic type *Cell*.

If the stack removal is not possible in a function a stack architecture is kept but with a Java API. The generated code looks like:

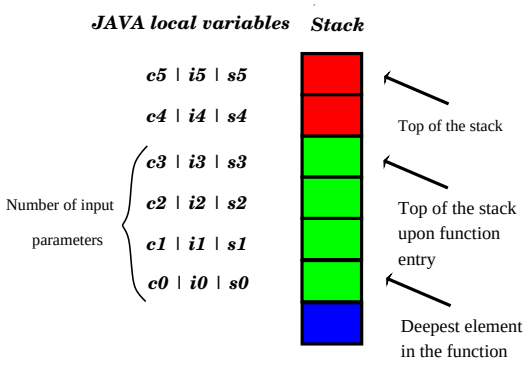

Figure 7: Variable notations in BiSTrO

```
public void format_bdate()
{
  stack.push(year);
  stack.push(BuiltIn.empty(
         stack.pop().getString()));
  if( stack.pop().getInt() > 0)
  {
    stack.push("there's no year in ");
    stack.push(BuiltIn.cite( bib ));
    stack.push(stack.pop().getString()
              +stack.pop().getString());
    System.err.println("Warning : "
                       +stack.pop()
                             .getString()
                       +".");
  }
  else
  {
    stack.push(year);
  }
}
```

If a function with a stack is called by another function, this one will have a stack too.

All the calls to any of the 37 built-in functions of BibTeX are translated to direct Java code when possible (such as for +) or to calls to equivalent functions that has been rewritten in the BibTeX++ library.

For example, the translation of this bst function:

```
FUNCTION {format.lastchecked}
{ lastchecked empty$
    { "" }
    { inbrackets "cited " lastchecked * }
  if$
}
```

will be (without optimizations)

```
public String format_lastchecked( )
```

```
{
   String s0 , s1;
   int i0;
   s0 = lastchecked;
   i0 = BuiltIn.empty( s0 );
   if( i0 > 0 )
   {
      s0 = "";
   }
   else
   {
      inbrackets( );
      s0 = "cited ";
      s1 = lastchecked;
      s0 = s0 + s1;
   }
   return( s0 );
}
```

We can see the call to the `empty` function and the translation of the `*` BST concatenation operator to the `+` JAVA concatenation.

The BST control flow operators `if$` and `while$` are replaced by their JAVA counterparts.

Just after these four passes, we have a JAVA AST but this code is not optimized at all, so we need to clarify it a little. It is why we use an optimizer further.

JAVA optimization We decided to use several types of optimizations which are independent in order to provide a final optimization which is fully customable by the user. But we do not need very complex optimizations because the aim of this is to increase the legibility and not speed the execution. Another reason for using simple optimizations is that the input code was written by human programmers in a not very understandable language, so they tried to write this code cleanly.

We just use eight different functions in order to do that:

an `if` optimizer just removes all the empty `then` or `else` blocks found in a plain BST code;

a copy and constant propagation is used to remove the need of most of the variables generated instead of the stack usage. It will increase the quality of the next dead code elimination phase;

a dead code elimination is associated with the propagation optimization to remove many useless definitions, because it deletes all *write after write* dependencies;

a boolean translator: since in the BST language there is no boolean type, this optimization tries to transform integer into boolean in `if` and `while` conditions. For exemple, it will transform `if( BuiltIn.equal( i1 , i2 ) > 0 )` to `if( i1 == i2 )`

some other small optimizations like peep-hole optimization and poor-man partial evaluation: transforming `a1 = a0 + 0;` into `a1 = a0;` or `"some"+"thing"` into `"something"`,... These optimizations only try to increase the readability of the JAVA code.

After these optimizations we get some cleaner code such as for a previous example:

```
public String format_lastchecked( )
{
   String s0;
   if( BuiltIn.empty( lastchecked ) > 0 )
   {
      s0 = "";
   }
   else
   {
      inbrackets( );
      s0 = "cited " + lastchecked;
   }
   return( s0 );
}
```

If we look at another function from `plain.bst`:

```
FUNCTION {new.sentence}
{ output.state after.block =
   'skip$
   { output.state before.all =
      'skip$
      { after.sentence 'output.state := }
     if$
   }
  if$
}
```

it is translated to a rather long JAVA code:

```
public void new_sentence( )
{
   int i0 , i1;
   i0 = output_state;
   i1 = after_block;
   i0 = BuiltIn.equal( i1 , i0 );
   if( i0 > 0 )
   {
   }
   else
   {
      i0 = output_state;
      i1 = before_all;
      i0 = BuiltIn.equal( i1 , i0 );
      if( i0 > 0 )
      {
```

```
      }
      else
      {
         i0 = after_sentence;
         output_state = i0;
      }
   }
}
```

but the optimizations will downsize it to a more understandable function:

```
public void new_sentence( )
{
   if( output_state != after_block )
   {
      if( output_state != before_all )
      {
         output_state = after_sentence;
      }
   }
}
```

We can see in this example many different optimizations at work:

- all empty *then* blocks have been removed;
- all the global variables have been propagated: the `i0 = after_sentence; output_state = i0;` have been transformed into `output_state = after_sentence;`. We do not use longer any local variable;
- some built-in functions have been converted to boolean operators: the `BuiltIn.equal( i1 , i0 ) > 0` is now a simple `i1 > i0`.

So we can see here that all these transformations are pretty efficient. Indeed, the optimized function is much more readable that the non-optimized one.

Plugins and meta-plugins

Plugins architecture The strategy pattern used is based on hook mechanisms such as the one used in the Emacs editor. In the original design, many hook points are chosen to able the user to insert calls to her(his) own function.

From a software engineering point of view, it is close to aspect programming but in a more restrictive way since all the points that can be modified are defined in advance. We think this approach is more tractable but if some features are not easy to implement with the existing hooks, other can be added since BibTeX++ is also an evolving open source program.

Lots of hooks objects can be used to replace some objects in the current architecture to modify the global behavior as in the following examples.

Some other specialization frameworks remain to be studied further in this context to fit future extensions, such as direct subclassing of BibTeX++ classes, aspect programming, reflection and introspection on BibTeX++ class. The main issue is that the code complexity remains manageable and the security is not endangered.

Parsers The management of the various inputs is dealt by some parsers crafted to each input format. They rely heavily on SableCC the parser generator (Gagnon, 1998) to speed up the retargeting of BibTeX++ to a new data format.

Some new parsers can be loaded as plugins in BibTeX++ when requested by the user.

Prettyprinters Writing some plugins to output new `.bib` database files does not cause any trouble since it is a simple pretty-printer that outputs the internal database representation.

But translating automatically the BibTeX++ output into another language or targeting a new typesetting system is far more challenging. Basically, BibTeX++ is a tool able to run BibTeX++ native code or BibTeX code in an improved way but is not able to abstract the semantics of what a BibTeX code itself really does of course (in fact it is an intractable issue from the computer science). BibTeX++ has no idea about the fact it is outputting an author name or a conference name: it is executing a Java procedure with a side effect that outputs a string.

Thus, we can only rely on some heuristics to deal with the prettyprinter parameterization, such as recognizing a `.bib` text output and translating it.

Since it is not possible to understand the BibTeX++ style it is not possible to modify it for retargeting the LaTeX output to another typesetting format. Instead, one can translate the LaTeX output item to another format. It is simple to do since generally bibliographical styles do not generate complex TeX programs but only some text with simple LaTeX mark-up tags. On the other hand, the *key* information being directly managed by BibTeX and BibTeX++ can be dealt by the plugin and output to the correct format.

Modifying the output of the bibliographical style with a plugin is harder since one should access the type of the data. For example if one wants to write a plugin to modify any existing style to display the dates in a numerical form instead of a textual one for example by replacing with regular expression mechanisms all the occurrence to "November" with 11, one needs to apply this translation process

only on the date field we are... not aware of! What would happen if an author is named "November"?

This kind of problems is similar to automatic translation of buggy pre-year-2000 programs that cannot deal with years after 1999 to cleaner programs that can deal with them. Automatic transformations must be applied only on code that certainly deal with some dates and no other numerical computations.

An interesting approach could be to type the output text with the data attribute used to built this text through BibTeX++, in order to approximate the data dependance graph with a slicing approach (Weiser, 1984) (that is to extract from the style code only the minimal code needed to generate the value of a given variable) statically during the BibTeX compilation process or by statically analyzing the Java BibTeX++ style. In this way it could be easy to determine that this part of the text is a textual representation of the year, the author names or the title and use information to translate these texts in a representation without `\bibitem` and so on suitable for other typesetting tools. The typeset output for LaTeX could thus be retargeted easily since we would now in the output what is an author name, what is a surname, what is a conference title, etc.

Since we only want to know what input fields are involved on a given output field, one can use dynamic dependence graph reconstruction. Since BibTeX++ is programmed in an object oriented language, overloading of the data type class to embed this on-the-fly dependence graph construction could be easy. At the BibTeX++ output procedure, for each character, the input fields used to compute its value is known.

It is similar for example to the tainting concept of variables used in the perl language for security reasons, to know if a variable value has been computed by using a value given by the user or not. If yes and that the variable is used to execute a privileged operation, the programmer may refuse to execute such a dangerous thing by using the tainted mode.

The automatic internationalization process use the same approach to translate the output text from one language to another one. But if we have already bibliographical styles, say, for l languages and we want to be able to translate every language to any other, we need to write $l(l-1)$ translators, which is cumbersome. Instead, if we introduce a kind of "*esperanto*" intermediate abstract representation, we only need to write l translators to this intermediate form and only l pretty-printers to each language.

An other way for classical bst code crude translation could be to use pattern matching to translate common piece of code. Of course, since no semantics recognition would be used, this method is not very adaptable.

Code transformation It is interesting to have a code transformation engine in BibTeX++ to allow some plugins to modify the behavior of other styles or plugins to offer the feature envisioned in section 'Code transformation'.

Transformations could be done at different levels in BibTeX++, on the bst internal representation or the Java BibTeX++ style more generally, or more conservatively on the input data structures.

The code transformation itself could use hook mechanisms already present in the code or by using an aspect programming tier in the BibTeX++ infrastructure. A low level approach could be to allow plugins to modify the code (for example adding at the end of the output routine a call to a new procedure that will output a new field) by using Java reflection and introspection

Of course, from the security point of view this should be very carefully controlled to avoid for example some plugin malware to rewrite some security check parts of BibTeX++. But this can also be enforced by the underlying Java security model (§ 'Security model').

Security model In such a tool where code can be downloaded by foreign documents automatically from alien servers transparently, security sounds a little scary and it could be easy to design BibTeX++ viruses.

Hopefully BibTeX++ is written in Java that implements a sensible security model to control in a centralized way the execution of arbitrary code (JAVASEC, 2003) at a very fine level.

We use this mechanism that has been tailored to allow secure execution of programs retrieved from the Internet (applets) in www browsers in a similar way.

It is used in BibTeX++ to avoid the plugin code to access local sensitive resources, to avoid a plugin to modify the security infrastructure of BibTeX++,...

At the installation the `policy` file contains

```
grant {
permission java.util.PropertyPermission
  "bib_max", "read,write";
};
```

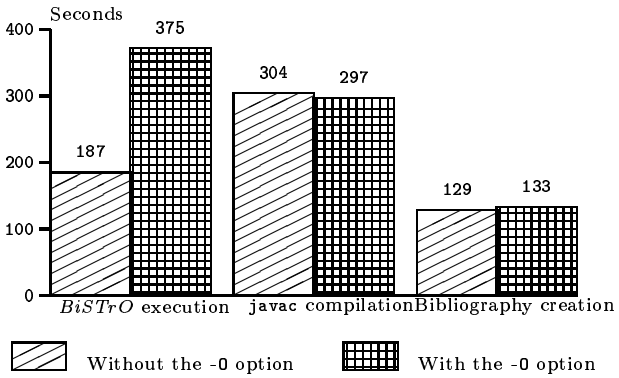

Figure 8: Total execution time of the BibTeX++ components on 152 styles.

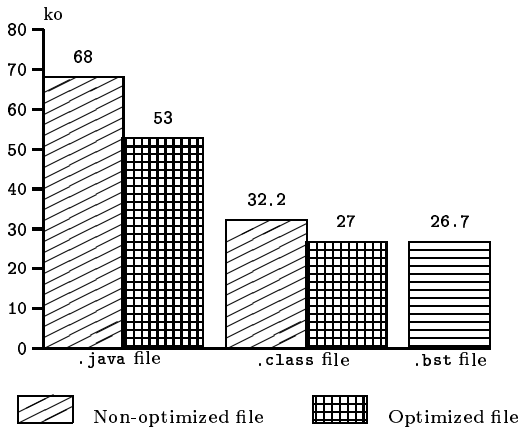

Figure 9: Mean size of a some style formats.

```
grant codeBase "file:${bib_lib}" {
permission java.io.FilePermission
  "<<ALL FILES>>", "read,write,execute";
permission java.util.PropertyPermission
  "bib_cache", "read";
permission java.util.PropertyPermission
  "bib_lib", "read";
permission java.util.PropertyPermission
  "java.class.path", "read";
};
```

All classes but those inside the BibTeX++ library (so only plugins and styles) can only access one environnement variable: `bib_max`. It is the maximum length of a string in a style.

Performance results With the power of modern computers compared with the older computers at the begin of BibTeX we may think that compiling a bibliography must be quite fast and not significant in the composing time.

But if BibTeX is quite simple and fast, BibTeX++ is quite more complex with its compiler engine and various optimization phases and one could ask if it is still fast enough.

Some tests were made on a Athlon XP 2000+ PC with 512 MB of RAM with *j2re* (Java 2 Runtime Environment) version 1.4.1 for Linux. The BibTeX++ programs were compiled by the Sun `javac` with the `-O` option to optimize the code.

Besides the performance evaluation, the compatibility of BiSTrO with BibTeX styles has been investigated by compiling all the 152 styles of the MikTeXdistribution. It allowed us to find and correct some bugs in our software but also in some other old BibTeX styles.

The execution time and the size of the styles between the optimized version of BiSTrO and BibTeX++ and the normal one has been compared and on figure 8, we can see that it is two times slower to optimize the Java style file. Nevertheless, the compilation of a `.bst` style file to a Java class file remains quite fast: only 3.2 seconds without any optimization and 4.4 seconds with all of them. Furthermore, since we use a cache mechanism to compile a new BST file to Java only one, this compilation time is spent only the first time we use a BST style: all the subsequent executions of BibTeX++ with an old BibTeX style will skip this compilation phase an run faster.

The execution of BibTeX++ is far slower than the original BibTeX (that runs in around 0.04 second on a small example), but this is not disturbing for a normal user because he just has to compile once the `.bst` file (it takes 3.2 seconds) and next time the creation of the `.bbl` file will only take 0.8 second.

Finally, the optimizations are not very useful for a standard BibTeX++ user: it increases the compilation time but does not decrease the execution time.

Nevertheless they are useful for style designers. Indeed we have already seen that there are two ways for creating a new style, by modifying an already existing one or by creating a new one from scratch. For both, it is easier with BibTeX++ than with BibTeX because the Java language is far more expressive, higher level and comprehensible than the BST language.

In order to help developpers who want to reuse an old style, they can ask BiSTrO to optimize its output code. Nevertheless, as we have just seen, these optimizations do not decrease the execution time of the bibliography generation, mainly because the Java compiler also optimizes the code when we compile it. So these optimizations are mainly useful for a style designer as a reverse-engineering framework.

We can see in the histogram in the figure 9 that these optimizations decrease the size of the `.java` style file by 22% and the size of the `.class` file by 16%. So they reduce the length of the code at least (it is often easier to understand a smaller code) but also increase its legibility as we have seen in the optimization code examples.

We can also note that the optimized `.class` files has almost the same size than the original `.bst` files. But since these class files were automatically generated we can imagine that handmade BibTeX++ style files will be smaller than BibTeX ones, so they will be easily downloadable and sharable.

Related work

There are many other open standard tools available to deal with bibliographical notices (D'Arcus and Lee, 2003) but at our knowledge none tackles both extensibility and BibTeX compatibility.

Nevertheless some are more related to our project, such as MLBibTeX and Bibulus.

MLBibTeX MLBibTeX (Hufflen, 2003) is a multilingual version of BibTeX rewritten in C. The database files and the behavior remains mostly compatible with BibTeX with small extensions.

This article introduces in a well documented way the domain and the issues related with bibliography internationalization and typography

The main point of MLBibTeX is the introduction of language switches inside the bibliographical items to be able to choose the most correct translation given by the author according to the current language, such as different notes or different transliteration of an author name.

Some other fields, such as the dates and so on are also naturally translated.

Some extensions are planned, such as using UNICODE and a localized sort but extensibility relies on adding new things in the code.

Bibulus An other tool tackles multilingual bibliographies with an extensible framework but without a BibTeX compatibility from the style point of view: Bibulus (Widmann, 2003).

As for BibTeX++ written in Java, Bibulus is written in a language that deal natively with UNICODE, PERL. This allow to deal with all the world languages. Since collators are also available, sorting is done according the requested language.

The input database format use XML but a tool has been written to translate BibTeX native `.bib` files to Bibulus one. The format in more typed to ease further internationalization. For example the gender of the author is defined to allow some grammatical variation in some languages.

Even if Bibulus is right now targeted at LaTeX environment, other input or output formats could be easily added.

Some style parameters can be written directly in the source document to change the behavior and new items can be added to a citation to override or to specialize some points of the bibliography for this very document or add some annotation in the current context and language[3].

The extensibility is based on two methods. First, as in BibTeX++, there are many hooks that able the style programmer to modify the standard behavior of Bibulus, such as rewriting things in the parser and so on. Next, since PERL is also an object oriented language, the style programmer can override some methods of Bibulus objects.

But since there is no security model in PERL beyond the tainted mode, it seems difficult to allow a secure execution for styles from the hostile world.

Conclusion

BibTeX++ is an extendable tool to deal with the bibliographical area in electronic document. It aims at extending the well known BibTeX in the LaTeX world by adding modern features such as UNICODE document encoding, Internet capabilities, scalability to future usages and future tools through plugin mechanisms and in the same time to remain compatible with the plain old BibTeX.

BibTeX++ is a free software piece of code written in a clean portable object oriented language than natively deal with UNICODE. Since it is written in Java, BibTeX++ is ready to run on every Java-enabled computer, even the installation phase is still to be streamlined.

BibTeX++ use advanced compiler techniques in a compiler (BiSTro) to recycle legacy dusty deck BST and BIB files. It translates a native BibTeX style written in the BST stack language to a new BibTeX++ style written in Java than can be extended further as a basis of a class of new styles. Right now, BibTeX++ has been tested on Linux and Windows™ on all the BibTeX styles found in the teTeX and MikTeX distribution. Indeed it allowed us to find that some of these styles are incorrect.

The plugin architecture is still to be developed in the current version with a general extension framework.

[3] This interesting idea could be realized in BibTeX++ by writing a plugin.

Other input front-ends and output back-ends are still to write for other tools than LaTeX such as OpenOffice or DocBook. But once these plugins are written we can reach the great bibliographical unification: for example having a Word™ document with an XML bibliography database fetched from the Internet using a `.bst` BibTeX style for a journal found on the Internet.

The BiSTrO compiler that currently generate Java code for a Java BibTeX library could be retargeted towards other bibliographical tools such as Bibulus in PERL with its own library.

An other usage of BiSTrO is to ease the development of plain BibTeX files. Since it can translate BST code into cleaner Java code it can be seen as a reverse-engineering tool for people more comfortable with Java than with BST.

A code transformation framework for automatic localization of a BibTeX existing style is still to be studied to understand the output from a given language to another one.

It is fascinating from a computer science point of view to see how many interesting research questions are to be solved in a tool as simple at first glance as a bibliographical management system. But this must not move us away from the typography domain with some issues such as how to deal with complete mix up of Latin, Arabic, Chinese,... different entries in the *same* bibliography and so on.

Further information on BibTeX++ with its code can be found on `http://bibtex.enstb.org`.

Thanks

The authors want to thank all the students that have worked during their studies on the BibTeX++ project with them through various internships in the Computer Science Laboratory at ENSTBr : Laurent Cordival, Guillaume Ferrier, Emmanuel Valliet who programmed the first lines and the infrastructure with Nicolas Torneri during their first year internship (PAP 2P in 2000), Étienne de Benoist, Martin Brisbarre, Aude Jacquot, Olivier Muller, Mathieu Servillat and Mohamed Firass Squalli Houssaini who extended it (PAP 5J in 2001), Sergio Grau Puerto for the first review of stack removal.

References

Abrahamsen, Per and D. Kastrup. "AUCTeX: An integrated TeX/LaTeX environment". 2004. `http://www.gnu.org/software/auctex`.

Briggs, Preston. "Register Allocation via Graph Coloring". Technical Report TR92-183, Rice University, 1992.

Callahan, David and B. Koblenz. "Register allocation via hierarchical graph coloring". In *SIGPLAN 91 : Conference on Programming Language Design and Implementation*, pages 192–203. 1991.

Chow, Fred C. "Minimizing register usage penalty at procedure calls". In *SIGPLAN '88 : Conference on Programming Language Design and Implementation*, pages 45–58. 1988.

Daly, Patrick W. "CUSTOM-BIB Package". Max-Planck-Institut für Aeronomie, 2004. `http://www.ctan.org/tex-archive/macros/latex/contrib/custom-bib`.

D'Arcus, Bruce and J. J. Lee. "Open standards and software for bibliographies and cataloging". 2003. `http://wwwsearch.sourceforge.net/bib/openbib.html`.

Dominik, Carsten and S. Eglen. "RefTeX — Support for LaTeX Labels, References and Citations with GNU Emacs". 2004. `http://remote.science.uva.nl/~dominik/Tools/reftex`.

Donin de Rosière, Emmanuel. *From stack removing in stack-based languages to BibTeX++*. Diplôme d'étude approfondie, ENSTBr, 2003a. `http://www.lit.enstb.org/~keryell/eleves/ENSTBr/2002-2003/DEA/Donin_de_Rosiere`.

Donin de Rosière, Emmanuel. *État de l'art sur les logiciels de gestion de références bibliographiques compatibles avec LaTeX et sur la suppression de la pile dans les langages à pile.* Étude bibliographique de diplôme d'Étude approfondie, ENSTBr, 2003b. `http://www.lit.enstb.org/~keryell/eleves/ENSTBr/2002-2003/EB/Donin_de_Rosiere`.

Ertl, M. Anton. "A New Approach to Forth Native Code Generation". In *EuroForth '92*, pages 73–78. 1992.

Ertl, M. Anton. *Implementation of Stack-Based Languages on Register Machines.* Ph.D. thesis, Technische Universität Wien, 1996.

Gagnon, Étienne. *SableCC, an object-oriented compiler framework.* Ph.D. thesis, School of Computer Science, McGill University, Montreal, 1998.

Goossens, Michel, F. Mittelbach, and A. Samarin. *The LaTeX Companion.* Addison-Weslay, 1994.

Hufflen, Jean-Michel. "European Bibliography Styles and mlBibTeX". In *EuroTeX'2003 – 14th European TeX Conference*, ENST Bretagne, France. 2003.

INMOS. *The Transputer Databook.* Inmos, 1989.

ISO. *Référence bibliographiques : Éléments essentiels*, 1958.

JAVASEC. "Java Security". 2003. http://java.sun.com/security.

Keryell, Ronan. "Publication list". 2004. http://www.lit.enstb.org/~keryell/publications/biblio/html.

Larousse. *Le Petit Larousse Illustré*, volume 1. Larousse, 2002.

latex. "LaTeX — A document preparation system". http://www.latex-project.org, 2004.

Lawrence, Steve, C. L. Giles, and K. Bollacker. "Citeseer, The NEC Research Institute Scientific Literature Digital Library". 2002. http://citeseer.nj.nec.com.

Patashnik, Oren. *BibTeXing*, 1988.

Proebsting, Todd A. and S. A. Watterson. "Krakatoa : Decompilation in Java (Does Bytecode Reveal Source ?)". In *Third USENIX Conf. Object-Oriented Technologies and Systems (COOTS)*, pages 185–197. 1997.

Vliet, H.-P. V. "Mocha, java bytecode decompiler". 2003. http://www.brouhaha.com/~eric/computers/mocha.html.

Weiser, Mark. "Program slicing". *IEEE Transactions on Software Engineering* **10**(4), 352–357, 1984.

Widmann, Thomas. "Bibulus — a Perl/XML replacement for BibTeX". In *EuroTeX'2003 – 14th European TeX Conference*, ENST Bretagne, France. 2003.

Yang, Byung-Sun, S.-M. Moon, and E. R. Altman. "LaTTe: A Java VM Just-In-Time Compiler with Fast and Efficient Register Allocation". In *International Conference on Parallel Architectures and Compilation Techniques,*, pages 128–138. 1999.