BibTeX++: Toward Higher-order BibTeXing

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

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

Youssef Aoun, Laura Barrero Sastre, Emmanuel Donin de Rosière, Nicolas Torneri
(Emmanuel.DoninDeRosiere|Nicolas.Torneri)@enst-bretagne.fr

Abstract

In the LaTeX world, BibTeX is a very widely used tool dealing with bibliographies. Unfortunately, this tool has not evolved for the last 10 years and even if a 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 object oriented aspects and offers new functionalities: for the expressiveness, the security model and native Unicode character set support, BibTeX++ styles are directly Java classes; 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 into 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 extend BibTeX++ functionalities. For example, since (meta-)plugins can load new plugins, new styles can be directly downloaded from the Internet.

Résumé

Dans le monde LaTeX, BibTeX est un outil répandu pour gérer les notices bibliographiques. Malheureusement, cet outil n'a pas évolué ces dix dernières années. Même si d'autres outils bibliographiques existent, il semble intéressant de développer un nouvel outil, offrant de nouvelles fonctionnalités et utilisant de nouveaux standards de programmation.

Le projet BibTeX ++ a démarré en 1999 et il est écrit en Java pour des raisons de portabilité et de fonctionnalités accrues dues aux aspects de programmation orientée objet. Il offre de nouvelles fonctionnalités : BibTeX ++ est une classe Java, pour son expressivité, le modèle de sécurité et le support natif d'Unicode ; il contient un compilateur pour traduire les anciens styles BibTeX en Java, et ce code Java peut être à la base d'un style bibliographique BibTeX ++ natif de manière à faciliter la vie du programmeur ; l'architecture a été conçue en vue de l'extensibilité : le noyau, les parseurs (pour être compatible avec d'autres styles bibliographiques, d'autres sources ou codages), les enjoliveurs de code (pour générer des bibliographies pour d'autres outils que LaTeX), les plug-ins ; le concept de plug-in est utilisé pour étendre BibTeX ++ de manière dynamique. Ainsi, puisque les (meta-)plug-ins peut à leur tour charger d'autres plug-ins, des nouveaux styles peuvent être téléchargés sur 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. These references are used to:

- clearly identify 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 [17] (we often use the name of the book, the author, the editor, ...), but because of the digital revolution, there are more and more document types (web pages for example) and identification elements (e-mail, URL, ...). So the management of the references has become 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 a headache for typographers: if the article talks about "Larousse (2002)" as a book introducing metamiddleware [20] and in the bibliography "Larousse 2002" is described instead as a French cookbook, there has clearly been a problem somewhere. Another problem is that each journal has its own bibliography style, so bibliographical management software has to allow the user to create his own style and send it to other people.

Fortunately, BibTeX [23] is a popular tool used by the LaTeX [21] community to generate bibliographical notices in publications. If there are also many other tools available [9], this one suits very well the LaTeX philosophy and is well integrated with 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 made by BibTeX to the `.bbl` file by using a `.bst` bibliography style file. The work-flow is summed up in figure 1 [13].

There are a lot of bibliographical styles available for BibTeX targeted at many different scientific journals, book styles, etc. 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 [22].

There are many tools targeted at easing the management of all these BibTeX files: database tools, editors, ... For example to write this article we used the grand Emacs multi-platform text editor that has various

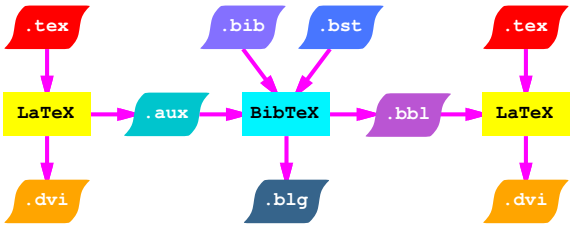

Fig. 1: BibTeX work flow.

modes to deal with a LaTeX document in various coding systems: the great AUCTeX mode [1] to deal with many things in the LaTeX source file, refTeX [7] 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 [22] and it is often easy to get some BibTeX records from few keywords.

Even though there is 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 an 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's brain. The tool lacks extensibility and such modern features as network awareness.

This 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 our BISTRO compiler (§ 'Software architecture of the BISTRO BST to JAVA compiler') and the plugin architecture (§ 'Plugins and metaplugins').

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. The ordinary BibTeX++ user does not have to embrace the object oriented architecture of BibTeX++, but this is not the case for the style programmer (and of course the BibTeX++ programmers).

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

BibTeX usage is simple; 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 being produced 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-Wesley},
      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) available 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 extensions 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 a `\cite{`⟨*key*⟩`}` command is 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 "[13]".

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 made by BibTeX to the ⟨*document*⟩`.bbl` file by using a ⟨*style*⟩`.bst` bibliography style file and the citation matter found in the `.bib` file(s). Error and information output goes to the ⟨*document*⟩`.blg` file. The work-flow is summed up in figure 1 [13].

BibTeX comes with some predefined styles but it is also possible to program other styles, for example to accept a `url` field to cite Internet information. All these styles are defined in `.bst` files that are used by BibTeX

```
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$
}
```

Fig. 2: bst sample code.

to generate the bibliography according to the work flow described in 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 was 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. bst code looks like that in 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 change only few details. Fortunately there are many styles available already that can fit most needs, and there are also some custom style generators [5].

Extensions to BibTeX A basic extension is to be able to deal with encodings other than 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.

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

A tool designed today could hardly escape the networked world, so BibTeX++ must have a way to access world-wide on-line bibliographical databases 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 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). From a programmer point of view it is yet another (less) cryptic language to learn that is still cumbersome. The expressiveness may not suit everyone's 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 use 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 solution is acceptable.

The next question is to select an 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 libraries to deal with all the modern programming ways (Unicode, Internet, ...), and widespread 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 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 language different than 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 would 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 more deeply 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 an inexperienced 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 corrupting 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 needs to regard the input syntax that BibTeX++ can accept. New parsers 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 [22] through the 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.

In an easier way, BibTeX++ could be used to translate a `.bib` file 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 revive some of the old-fashioned styles; for instance, adding new `url` records to old styles written before the Internet wave.

To make a bibliographical database or a researcher publication list [19] 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 in any style by using 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++, code that can also be downloaded by a plugin itself. Since it is a plugin that can fetch from the network other plugins it has been nicknamed 'meta-plugin' in BibTeX++.

One can embrace various future uses 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 referencing all the bibliographical databases available on the Internet. A plugin associated with 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 material 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 to this kind of mobile code in the wild.

Plugin syntax To remain close to the existing LaTeX and BibTeX interaction syntax we do not add new LaTeX macros but rely on the current ones, merely adding special keywords inside the `\bibliography` and `\bibliographystyle` macros.

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

To avoid conflicts with 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 wants to directly use citations from the CiteSeer database [22], this will be chosen with

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

If a user wants to add a plugin style to automatically add the output of a `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 attributes with

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

Plugins themselves 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 a 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 in 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 been 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 clashes 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 equally well in either the `\bibliography` or `\bibliographystyle` macros, 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 stopping 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 be used with new macros, defined in a new package `bibtexpp`, if this compatibility is not mandatory.

Of course, a parser plugin can itself define a new plugin definition or parameter syntax to deal with other

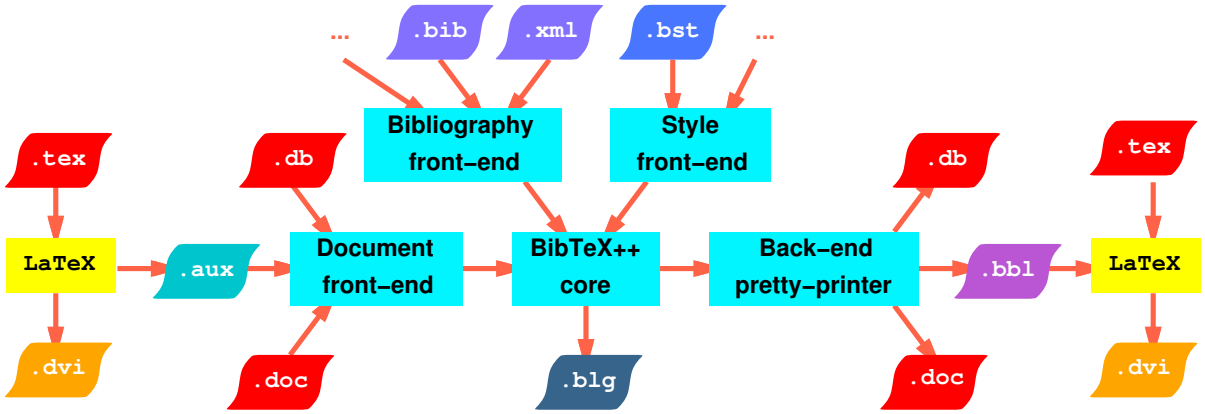

Fig. 3: BibTeX++ work-flow.

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 the bst language into something newer and more expressive.

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

To be more generic and more scalable, output is handled by prettyprinters of internal representations, and input is read by parsers that build a common internal representation. In this way, we have only to define a new prettyprinter or parser to deal with a new format, without having to change the BibTeX++ internals.

The parsers are made with Sablecc [12], a compiler compiler for Java written in Java. Unlike other compiler compilers (such as Java Cup, for example [14]), 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, database 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 a 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 common errors in bst files and optimize the 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 relatively simple part outputs 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 prettyprinter other output could be generated for another tool.

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

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

Bibliographical database front-end This front-end also follows a strategy pattern to cope with various database formats.

A Sablecc parser has been written now to read only BibTeX bibliographical database in the `.bib` format.

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

If the first access method is quite fast, the two other ways may be deadly slow compared to it. This 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 on every BibTeX++ run.

Software architecture of the bistro bst to Java compiler Now we consider the bistro compiler for translating bst to Java. See figure 4.

The BibTeX stack-based input language BibTeX++ `.bst` files use a stack-based language: it is a type of lan-

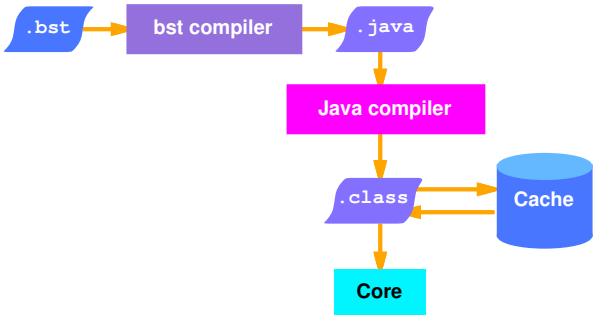

Fig. 4: Compilation work-flow.

guage where you push the data to manipulate on a stack. Also, functions store their results on this stack. The syntax uses a reverse polish notation (also called postfix notation). For example, to compute `2 + 3`, you first push `2` and `3` on the stack and then call the `add` operator, so you have to write something like this: `2 3 +`.

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

- Forth, used in many fields, especially embedded control applications;
- PostScript, used primarily in typesetting and other display purposes. 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 (BST) language, the stack-based language used for processing BibTeX databases.

Although stack-based languages are sometimes used as programming languages, they are more popular as intermediate languages for compilers 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 systems;
- the Smalltalk-80 bytecode is the intermediate language of the Smalltalk-80 system.

Some other stack-based languages target several machines as machine-independent languages, notably:

- the Java bytecode output from 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 languages are not frequently used. Today, there are 3 ways to execute a stack-based language on register machines, via:

- 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.

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 in alphabetical order using the author names, and italicize titles.

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

- global variables, declared by `INTEGERS` or `STRINGS` commands;
- 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.

Among the 10 BST commands available, here are some of the more interesting ones:

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

2. There is probably no other popular architecture since the Transputer [16].

All these commands support defining the structure of a style file. But with them, we can not manipulate variables. This 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$` which 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 (meaning "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 over-simplified example, it is not very easy to understand the BST language:

- we are not accustomed to 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.

This explains why only a few people are able to program a new style in this language. So for BibTeX++, we will have to create a more expressive style language. But because of the need for 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 languages A number of techniques have been proposed in the literature for this type of translation.

Source-to-source translator There are only a few source to source translators for stack-based language. The most famous research on this was done in [10, 11] where Forth code was translated into C in order to increase the portability of Forth applications. Ideally, 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* [24] or *mocha* [25] 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.

C variables	Stack	
x_1		↖
x_0		Top of the stack
p_0		↖
p_1		Top of the stack
p_2		upon function entry

FIG. 5: Stack mapping to C variables.

In [11] the *f2c* Forth to C translator described uses several steps to convert Forth code to C language. The first is to split the code into its functions which will be processed independently. Then, *f2c* counts the number of input and output parameters for each function. The next step is to convert the elements on the stack into C's local variables, as shown in figure 5.

In that figure, $p_0, p_1, \ldots$ represent the entry variables of each function and $x_0, x_1, \ldots$ 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 example, 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 useful 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. Since Forth allows the creation of arbitrary control structures, it is easiest to convert them into C `goto` instructions and labels.

This translation mechanism must know the height of the stack everywhere in the Forth code, but it is not always possible. Sometimes the stack depth is unknown. For example, the instruction `?DUP 0= IF` means that if the top of the stack is 0, replace it with the previous element on the stack, else we delete the element. So in this

case, *f2c* has to create a stack in C and use it throughout the whole function.

Compiler Source-to-source translators are not the only software 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 [10] 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.

After that, it converts the 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 to reduce register dependencies between instructions.

Now, we have a set of mnemonic blocks, but we have to connect them with 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 faster output code, three good register allocation algorithms are proposed: graph coloring register allocation [2], hierarchical graph coloring [3], and interprocedural allocators [4].

Another stack elimination in a compiler can be found in Java compilers. Today, faster and faster execution is needed for Java applications. Better Java performance 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 so as to improve the Java performance.

LaTTe [28] 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 assembler. 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.

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 variables used by LaTTe ;
- the remaining number distinguishes the symbolic registers.

For example, `il2` represents the second local integer register. 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 allocation using that information.

Sometimes, we need to move some registers in order to reconcile register allocation at region join points because LaTTe uses 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 the Sun JIT, and this speed comes particularly from the register allocation algorithm.

Compiling BibTeX BST styles 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. We planned a classical compiler architecture divided into several steps as shown in figure 6:

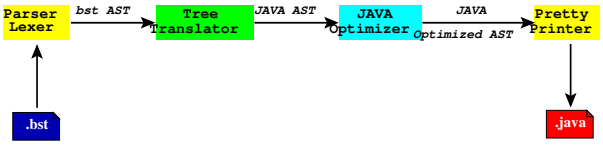

Fig. 6: bistro architecture.

- first, we use the bst parser to convert a bst file into a bst abstract syntax tree (ast) with a Sablecc grammar;
- then, in the tree translator the bst ast is transformed to a Java-like ast :
  - useful information from the tree is gathered, like function names, global variables, ... Functions are analyzed to decide if the stack can be removed and if so what is the number of input-output parameters;
  - unlike bst, Java is a type-based language, so we need to know the type of every variable in the non-stack-based session. But in some cases, it is very difficult to find this type, so if we cannot determine it, we use instead a *Cell* object: an object that can store both an integer and a string;
  - with this 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, a Java file is written by the prettyprinter 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 block of code has been optimized for human comprehension, it should 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 interpreter had been designed instead of a translator, this would not have been possible.

More information on the bst to Java compilation can be found in [8] but 2 phases are detailed here.

Java translation First, information about functions is searched for in 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, and 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 directions in a use-def or def-use way to deal with typed entries or output. Some further abstractions are used to follow the dependency graph even if there are stack manipulation operations 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 determine how many Java local variables we will have to use, and 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 when possible, or generate Java functions with a Java stack when stack removal is not possible. Variables are named accordingly to figure 7. If the type has been inferred, the native Java types `String` or `int` are used instead of our polymorphic type *Cell*.

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

```
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()
```

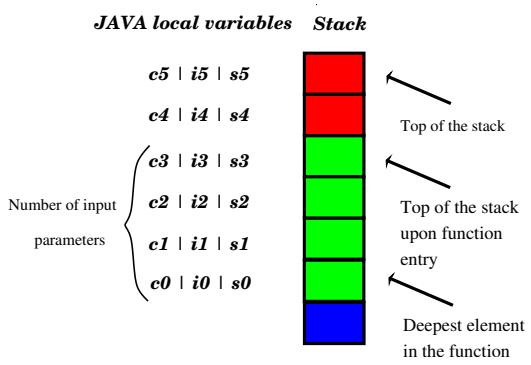

FIG. 7: Variable notations in BISTRO.

```
            +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, the latter will have a stack too.

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 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.

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.

JAVA optimization We decided to use several types of independent optimizations in order to provide a final optimization which is fully customizable by the user. We do not need very complex optimizations, because the aim of this is to increase legibility, rather than speed 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 use eight different functions for this:

- an `if` optimizer simply removes all the empty `then` or `else` blocks found in a plain BST code;
- a copy and constant propagation function removes most of the variables generated, instead of stack usage. This increases the quality of the next phase, dead code elimination;
- a dead code elimination function 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 to boolean in `if` and `while` conditions. For example, it will transform `if( BuiltIn.equal( i1 , i2 ) > 0 )` to `if( i1 == i2 );`
- other small optimizations such as peep-hole optimization and poor-man partial evaluation: for example, transforming `a1=a0+0;` into `a1=a0;` or `"some"+"thing"` into `"something"`, and so on. These optimizations only try to increase the readability of the JAVA code.

After these optimizations we get somewhat cleaner code, such as this from 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$
}
```

This is translated to 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 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 global variables have been propagated: the code `i0 = after_sentence; output_state = i0;` has been transformed into `output_state = after_sentence;` We no longer use any local variables;
- 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 that used in the Emacs editor. In the original design, many hook points are chosen to enable users to insert calls to their own functions.

From a software engineering point of view, it is close to aspect programming, but in a 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, more can be added since BibTeX++ is also an evolving open source program.

Lots of hook objects can be used to replace objects in the current architecture, thus modifying 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, and reflection and introspection on BibTeX++ classes. The main issue is that code complexity remains manageable and security is not endangered.

Parsers The management of the various inputs is dealt with by parsers crafted to each input format. They rely heavily on the Sablecc parser generator [12] to speed up retargeting of BibTeX++ to a new data format.

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

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

But automatically translating 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 piece of BibTeX code itself really does (of course; in fact, this is an intractable issue from theoretical computer science). BibTeX++ has no idea that 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. This is simple to do since generally bibliography styles do not generate complex TeX programs, but only text with some simple LaTeX mark-up tags. On the other hand, the *key* information, being directly managed by BibTeX and BibTeX++, can be dealt with by the plugin and output in the correct format.

Modifying the output of the bibliography 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, say by replacing via regular expression mechanisms all occurrences of "November" with "11", one needs to apply this translation process only on the date field which we are ... not aware of! And what would happen if an author is named "November"?

This kind of problem 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 deals with dates and not other numerical computations.

An interesting approach could be to type the output text with the data attribute used to build this text through BibTeX++, in order to approximate the data dependence graph with a slicing approach [26] (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 a given part of the text is a textual representation of the year, the author names, or the title, and use this 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 have 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, we 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.

This 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 variable is used to execute a privileged operation, the programmer may refuse to execute such a dangerous thing by using tainted mode.

The automatic internationalization process uses the same approach to translate the output text from one language to another. But if we have existing bibliography 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 then l prettyprinters to each language.

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

Code transformation It is interesting to have a code transformation engine in BibTeX++ to allow plugins to modify the behavior of other styles, and implement other features.

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 the hook mechanisms already present in the code, or 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 plugin malware rewriting security checking in BibTeX++. But this can also be enforced by the underlying Java security model, which we turn to now.

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

Hopefully, BibTeX++ is written in Java, which implements a sensible security model controlling in a centralized way the execution of arbitrary code [18] 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 web browsers in a similar way.

It is used in BibTeX++ to avoid plugin code accessing sensitive local resources, modifying the security infrastructure of BibTeX++, and other obvious concerns.

At installation, the `policy` file contains

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

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 except those inside the BibTeX++ library (thus, plugins and styles) can only access one environment 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 beginning of BibTeX, we may think that compiling a bibliography must be quite fast and not significant in the composing time.

But although the original BibTeX is quite simple and fast, BibTeX++ is noticeably more complex with its compiler engine and various optimization phases. 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 152 styles in the MiKTeX distribution. This allowed us to find and correct some bugs in our software, as well as some bugs in old BibTeX styles.

The execution time and the size of the styles between the optimized version of bistro and BibTeX++ and the normal one is compared in figure 8. We can see with the bistro execution time that it is two times slower to generate optimized Java style file than non-optimized one. Nevertheless, the compilation of a `.bst` style file to a Java class file remains quite fast, by considering the

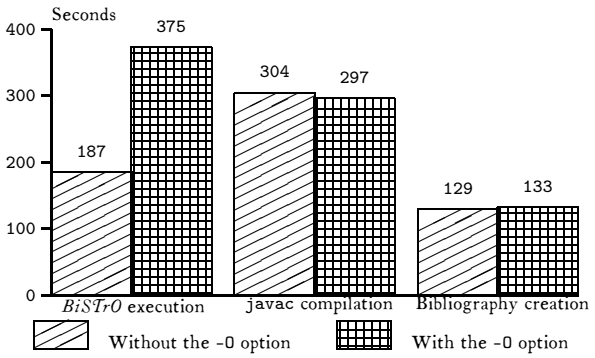

Fig. 8: Total execution time of the BibTeX++ components on 152 styles.

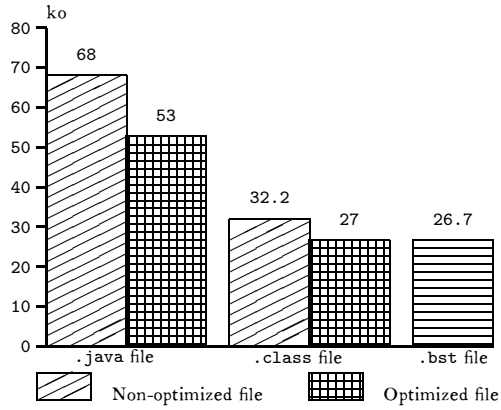

Fig. 9: Mean size of some style formats.

bistro execution time plus the `javac` execution time to generate the class file: only 3.2 seconds on average per `.bst` file without any optimization and 4.4 seconds with all the optimizations. Furthermore, since we use a cache mechanism to compile a new bst file to Java only once, this compilation time is spent only the first time we use a bst style: all subsequent executions of BibTeX++ with the old BibTeX style will skip the compilation phase and thus run faster.

The execution of BibTeX++ is far slower than the original BibTeX (that runs in about 0.04 second on a small example), but this is not disturbing for a normal user because he just has to compile the `.bst` file once (the 3.2 seconds above), 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 of creating a new style, by modifying an 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 developers 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 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 usually easier to understand shorter code), but also increases 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 as the original `.bst` files. But since these class files were automatically generated we can imagine that hand-made BIBTEX++ style files will be smaller than BIBTEX ones, so they will be more easily downloadable and sharable.

Related work

There are many other open tools available to deal with bibliographies [6], but to our knowledge none tackles both extensibility and BIBTEX compatibility.

Nevertheless some are closely related to our project, such as MlBIBTEX and Bibulus.

MlBIBTEX MlBIBTEX [15] is a multilingual version of BIBTEX rewritten in C. The database files and behavior remain mostly compatible with BIBTEX with small extensions.

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

The main point of MlBIBTEX is the introduction of language switches inside the bibliography 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 transliterations 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 features in the code.

Bibulus Another tool tackles multilingual bibliographies with an extensible framework, but without a BIBTEX compatibility for styles: Bibulus [27].

As with BIBTEX++ written in Java, Bibulus is written in a language that can deal natively with Unicode: Perl. This allows dealing with all the world's languages. Since collators are also available, sorting can be done according to the requested language.

The input database format uses XML but a tool has been written to translate BIBTEX's native `.bib` files to the Bibulus format. The format is typed more strongly, to ease further internationalization. For example the gender of the author is defined to allow for grammatical variation in some languages.

Although Bibulus is right now targeted at the LATEX environment, other input or output formats could easily be 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 specialize some points of the bibliography for this particular document, or to add an annotation in the current context and language.[3]

The extensibility is based on two methods. First, as in BIBTEX++, there are many hooks to enable 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 tainted mode, it seems difficult to allow for secure execution of styles from the hostile world.

Conclusion

BIBTEX++ is an extendable tool dealing with the bibliographical area of electronic documents. 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 at the same time to remain compatible with plain old BIBTEX.

BIBTEX++ is a free software program written in JAVA, a clean portable object oriented language that natively handles Unicode. Since it is written in JAVA, BIBTEX++ is ready to run on every JAVA-enabled computer, although the installation phase is still to be streamlined.

BIBTEX++ uses advanced compiler techniques in a compiler (BISTRO) for recycling "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 that 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.

3. This interesting idea could be realized in BIBTEX++ by writing a plugin.

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

Other input front-ends and output back-ends are still to be written for tools other than LaTeX, such as OpenOffice and 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 generates Java code for a Java BibTeX library could be retargeted to other bibliographical tools such as Bibulus in Perl with its own library.

Another 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 bst.

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

From a computer science point of view, it is fascinating 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 of Latin, Arabic, Chinese, ... entries in the *same* bibliography, and so on.

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

Thanks

The authors want to thank all the students that have worked with them during their studies on the BibTeX++ project through various internships in the Computer Science Laboratory at enstbr: Laurent Cordival, Guillaume Ferrier, and 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), and Sergio Grau Puerto for the first review of stack removal.

Bibliography

[1] Per Abrahamsen and David Kastrup. AUCTeX: An integrated TeX/LaTeX environment, 2004. `http://www.gnu.org/software/auctex`.

[2] Preston Briggs. Register allocation via graph coloring. Technical Report TR92-183, Rice University, 24, 1992.

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

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

[5] Patrick W. Daly. The custom-bib package. Max-Planck-Institut für Aeronomie, 2004. `http://www.ctan.org/tex-archive/macros/latex/contrib/custom-bib`.

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

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

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

[9] Emmanuel Donin de Rosière. 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, September 2003. `http://www.lit.enstb.org/~keryell/eleves/ENSTBr/2002-2003/EB/Donin_de_Rosiere`.

[10] M. Anton Ertl. A new approach to Forth native code generation. In *EuroForth '92*, pages 73–78, 1992.

[11] M. Anton Ertl. *Implementation of Stack-Based Languages on Register Machines*. PhD thesis, Technische Universität Wien, 1996.

[12] Étienne Gagnon. *SableCC, an object-oriented compiler framework*. PhD thesis, School of Computer Science, McGill University, Montreal, 1998.

[13] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, 1994.

[14] Scott E. Hudson. *CUP User's Manual*. Georgia Institute of Technology, March 1996.

[15] Jean-Michel Hufflen. European Bibliography Styles and MlBibTeX. In *EuroTeX 2003* (this volume), ENST Bretagne, France, June 2003.

[16] Inmos. *The Transputer Databook*, 1989.

[17] ISO. *ISO R77: Référence bibliographiques : Éléments essentiels*, 1958.

[18] Java security, 2003. http://java.sun.com/security.

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

[20] Larousse. *Le Petit Larousse Illustré*, volume 1. Larousse, 2002.

[21] LaTeX — a document preparation system. http://www.latex-project.org, 2004.

[22] Steve Lawrence, C. Lee Giles, and Kurt Bollacker. Citeseer, the NEC research institute scientific literature digital library, 2002. http://citeseer.nj.nec.com.

[23] Oren Patashnik. *BibTeXing*, 1988.

[24] Todd A. Proebsting and Scott A. Watterson. Krakatoa: Decompilation in Java (does bytecode reveal source?). In *Third USENIX Conf. Object-Oriented Technologies and Systems (COOTS)*, pages 185–197, 1997.

[25] H.-P. V. Vliet. Mocha, Java bytecode decompiler, 2003. http://www.brouhaha.com/~eric/computers/mocha.html.

[26] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[27] Thomas Widmann. Bibulus — a Perl/XML replacement for BibTeX. In *EuroTeX 2003* (this volume), ENST Bretagne, France, June 2003.

[28] Byung-Sun Yang, Soo-Mook Moon, and Erik 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.