Literature programming – tool support for authoring and reviewing a scientific paper

Yijun Yu[1], Markus Strohmaier[2], Greg McArthur[2], Jianguo Lu[3], John Mylopoulos[2]

[1]The Open University, UK
[2] University of Toronto, Canada
[3] University of Windsor, Canada

Abstract. Authoring or reviewing a scientific paper is tedious to avoid or to locate presentational errors. Errors such as spelling, grammar can be checked by existing tools, whereas structural errors for concepts are harder to detect. Converting a technical paper into a program, the "literature programming" proposed in this paper allows existing program analysis tools to be reused for detecting and resolving some of its writing problems. For example, a simple C/C++ parser can be reused to check type errors that can not be captured by spelling and grammatical checkers; redundancies and false dependencies can be exposed and removed by the restructuring tool we developed for C/C++ programs. In general, an analogy between the literature (in terms of papers) and the software (in terms of programs) is made to reuse software engineering tools for literature programming (writing/reviewing/studying). The work has been applied to a recently published ICSE paper, showing a promising direction of software engineering-aided literature programming.
Keywords knowledge management, TEX/ LATEX, C/C++, markup

1 Introduction

Reviewing a technical paper is tedious, if not painful: given limited time, a reviewer is required to understand the technical content of the work, to identify its novelty, as well as to check its writing errors in spelling, grammar and structure. In a prominent technical conference such as ICSE, a program committee member needs to review more than 20 papers in less than 3 months, on top of their own challenging regular research.

Writing a technical paper that can be accepted by the peer reviewers, on the other hand, is even more tedious. Aside of the creative work, a mundane and unavoidable task is to present the work into a human understandable form. For a technical conference, such form must follow certain structures in order to save the readers (including reviewers) time in getting the idea across.

Experience and skills may help speed up presenting the work with a high quality. Yet, it is still not uncommon that avoidable errors escape the critical eyes of authors and reviewers and finally creep into the proceedings (See examples in Section 4).

It is believed that having a mechanical tool to detect all the errors made by human is hard, as hard as passing a Turing test [1]. Yet, many tools such as spell-checkers, grammar-checkers, are so prevalently used by us because they can catch low-level errors that people tend to make, regardless how good their command of the language is.

In this paper, we propose a literature programming methodology to reuse programming tools in such literature tasks. We do not aim at solving the mission impossible – detecting all human-made errors. Rather, we aim at a more feasible task – reducing the time of a reader by catching structural errors that can interrupt the flow of readings. We show how simple programming tools such as C/C++ parser and header restructuring tools [2] can be reused for this purpose.

Related to our proposal not only in technology but also in terminology, *literate programming* was proposed to unify document and code into a single source [3–5]. Applying a literate programming tool, the source can be split into a document with exact code snippets and a program with well-documented comments. To differentiate our intention from that, *literature programming* facilitates the reuse of existing programming tools on literature.

The remainder of the paper is structured as follows. Section 2 presents a list of common writing errors that can be detected by programming tools; Section 3 presents a list of markups that can be used in deconstructing the paper into a C/C++ program; Section 4 shows the results of C/C++ parser and header restructuring tool on the equivalent programs of the papers; Section 5 discusses the results and sheds lights on further studies. The related work is presented in Section 6 followed by our conclusion in Section 7.

2 Common writing errors

In this section, we explain some common patterns of errors [6] that can be detected by our initial literature programming tool. These patterns are explained using a similar format of a design pattern language [7].

Rule 1. Undefined usage. A concept is defined after its first use.
Patterns. A definition of a concept is usually signaled by (i.e. occurred as) a predicate, or by an emphasis. A use of a concept is however, not necessary signaled. The name of the use and definition does not have to be expressed exactly as the same string: aliases and abbreviations can also be considered for the same concept.
Exceptions. A use without definition in abstract is OK: there a concept can be informally defined before it is formally defined.

Rule 2. Redundancies A concept is defined more than once.
Patterns. Two signaled definitions occur in the paper. The name of the re-definition does not have to be exactly the same string: aliases, abbreviations etc. are also considered the same.
Exceptions. A concept can be inexplicitly defined in the introduction before it is explicitly defined.

Rule 3. Unnecessary definitions. A definition is not needed.
Patterns. Given the major contribution (i.e. new concepts), a definition is not directly or indirectly dependent by any new concept. Or the defined concept is common-sense to the domain.
Exceptions. With an ontology, it is commonly agreed on which concept belongs to common sense in a domain.

Rule 4. False citations. An undefined concept is not imported from the cited source.
Patterns. The cited papers are converted into interfaces (e.g. headers) that define the contributions as exported concepts. Problem occurs when the expected concepts (as being used in the sentence of the citation) is not in the interface exposed by the cited paper.

Rule 5. Deep citations An undefined concept is imported from an indirectly cited source.
Patterns. The concept is not defined in the cited papers directly, but it is imported indirectly by the cited papers. Due to the limitation of human readers, if a concept can not be found in 2 levels of the citations, then we consider it to be a deep citation.

3 Deconstructing a paper

In our work, we draw an analogy between a paper and a C/C++ program such that it is easy to see how to deconstruct a paper and generate an object-oriented program that captures the structural relations among concepts. Table 1 summarizes the analogy of concepts comparing the world of papers with the world of programming, or software engineering in a broader sense.

A concept in a paper is related to a symbol in a C/C++ program. It is introduced into the paper either through a definition or a declaration (i.e. import). Similarly in a C/C++ program, a symbol needs to be defined, or declared such that the compiler can locate the definition of the symbol in related files.

In a paper, locating the original definition of a concept can be regarded as a citation of a related paper where the concept were originally defined. If the definition is not defined in any related papers, it has to be either a commonly understood concept that one assumes to have in the mind of the readers, or it has to be defined explicitly in the previous part of the paper, assuming the reader is reading it from the beginning to the end. The analogy of citing a paper can be mapped into including a header which contains the declarations of the cited concept symbols. Moreover, a paper can prepare such an interface to the paper that cites it into a header, exporting contributions or new concepts defined.

Besides syntactical relations among the definition and reuse of symbols, we can also reuse other types of relations in a C/C++ program to capture similar conceptual relations among concepts in a paper. For example, the specialization and generalization of concepts can be easily mapped into inheritance relations of objects, whereas the aggregation or division of concepts can be mapped to containment relations between objects.

Table 1. Analogy between a paper and a C/C++ program

paper	program	context
Concept	Symbol	C
Definition	Definition	C
Use	Use	
Citation	Inclusion	
Cited paper	Header interface (imported)	
Contributions	Header Interface (prototype)	
Generalization	Generalization	C++
Specialization	Specialization	

3.1 Prepare markup

The markup in Table 2 can be used when a human reader is annotating a paper in ink. To process them semi-automatically, one can define them using XML markup if the paper is written in Word or DocBook [8]. If it is written in LaTeX, one can define a set of macros as a reusable package to markup the above signals. In our experiment, we did the latter in hope that the literature programming tool can be connected to the existing literate programming tools for LaTeX. For papers not available in LaTeX, we first use the Acrobat tool to extract ASCII text from the PDF files, adding a few section breaks, then applies our the markup directly on the text.

Table 2. Markup on a paper

relation	TeX macros	C/C++ programs
Import	\import{Concept}	`#include "<jobname><nimport>.h"`
Export	\export{Concept}	`// <jobname>.h`
Define	\define{Concept}	`struct <Concept> { int dummy;}`
Use	\use{Concept}	`<Concept> concept<nuse>;`
Inherits	\inherit{Concept}{Super}	`<Concept>: <Super>;`
Contains	\inherit{Concept}{Sub}	`<Concept> { <Sub> sub;}`

In [9], such concept extraction has been studied using a software technology (TXL) on a web-site in a particular domain. It reported that 70% of the concepts identified by the TXL parser are correct, in comparison to the markup conducted by a human domain expert. In other words, if one applies the similar technique, we estimate that a similar reduction in effort can be achieved.

3.2 Generating a C/C++ program

Our markup macros can be ignored when typesetting a paper. However, they are used in generating a C/C++ program. Meanwhile, the presented content of the paper is omitted when the program is produced. This is a technique used in literate programming [3]. The major differences are that (1) the resulting program must follow the lexical ordering of the paper and (2) the generated program is not meant for execution, rather, they are meant for program analysis.

Application of the literature programming macros A typical usage of such macro processing is explained as follows. A LaTeX writer can add a package declaration at the beginning of the document:

```
\documentclass{article}
\usepackage{lp}
```

Then inside the `document` environment, the macros can be used to markup the normal text, as illustrated by the following examples[1], respectively for *defining*, *using*, *importing* and *exporting* concepts.

```
\begin{document}
\use{FooBar}{foo bar is used}    \define{FooBar}{blabla is defined}
\import{FooBar}{... \cite{..}}    \export{FooBar}{...}
\end{document}
```

Macro `define` marks up a "definition" of the concept around the text that defines it. The name of the concept will be translated into a C/C++ identifier, therefore one must make sure it is already a valid alphanumeric string, to make this proof of concept work. In future we may tweak the TeX macro to allow more freedom in the concept names. Currently a user can use a LaTeX comment followed by the % sign to explain the concept if necessary. The second argument to the `define` macro is the text being marked. When a LaTeX processor is invoked, these text will be expanded as is. Nothing to add or to remove. In order to visualize the markups in the output, however, optionally we may highlight the text in a color to distinguish the role of the text in the concept dissemination. In the implemented `lp` (Literature Programming) package[2], we decorate a concept use in green, a concept define in red, a concept import in blue and a concept export in yellow.

The side effects of a LP macro, on the other hand, generate a C/C++ statement in the output program. A concept definition will become a structure definition in a generated header unit (.h file), a concept usage will become a variable definition using the concept structure in the compilation unit (.c file), a concept import will become an include statement in the compilation unit and a concept export will become an interface header unit which only include the necessary variable declarations.

Code generation Let us take the implementation for a concept usage macro for example. The following code snippet shows the implementation of the concept use macro by our program `lp.sty` in the TeX language [11].

```
1 \newcommand{\printuse}[1]{\addtocounter{usages}{1}
2 \immediate\write 0 {struct #1 concept\arabic{usages};}}
3 \newcommand{\use}[2]{\printuse{#1}\color{green}#2 \color{black}}
```

The \use macro (Line 3) takes two arguments. The first one is an identifier to the concept and the second one is the text being marked up. The body of the

[1] Larger examples of the marked up LaTeX document [10] can be seen at http://www.cs.toronto.edu/~ yijun/work/lp/p59-rajan-markedup.tex.txt as well as this paper http://www.cs.toronto.edu/~yijun/work/lp/lp.tex

[2] http://code.google.com/p/literature-programming

macro will expand it into two parts: The marked text ('#2') is highlighted in the generated paper using a special color – green – to signal a usage of a concept right in the human readable form. The \use macro definition calls for another macro expansion to output a statement for the C/C++ program using the 1st argument, specifically, a declaration statement for a variable. For example:

```
struct Concept concept2;
```

The generated variable has a unique name, which is guaranteed by a global counter of the times a concept is used. The variable also has a type "struct #1" where '#1' stands for the first argument of the \use macro. For this to work, two specific features in TEX are employed initially in the literature programming LATEX package. First, we need to initialize the global counter to zero to keep track of the number of uses in the paper (Line 1). Second, we need to initialize a number for auxiliary output streams (e.g. 0) for the C/C++ output program to be generated (Line 2). In the current implementation, we generate the main program as a compilation unit (`<jobname>.c`). Here "jobname" is the placeholder for the name of the LATEX job.

```
\newcounter{usages} \setcounter{usages}{0} \openout 0 \jobname.c
```

For the definition of a concept, we also introduced a macro, implemented as follows:

```
1 \newcommand{\printdef}[1]{\immediate\write 0 {struct #1{int dummy;};}}
2 \newcommand{\define}[2]{\printdef{#1}{\color{red} #2 \color{black}}}
```

The implementation of the definition macro (Line 4-7) is similar to the usage macro, using a different color, red, to indicate the concept definition in the paper. The main difference between this macro and the usage macro is in the C/C++ output: Instead of a variable declaration for the usage of the type, the generated statement will define the concept using '#1'. That's why the first argument must be a valid C identifier to make the generation sound. In order to make sure the type generated is a valid definition, we introduced a dummy field in the struct definition. An example C/C++ output from this macro is shown as follows:

```
struct Foo { int dummy; }
```

Having the definition and usage of concepts generated, we need to organize the interfaces between papers through the generated headers. That is, we generate a header for the exported concepts and a number of headers from the imported concepts. Moreover, we also generate the inclusion statements for including the imported concepts to the main program. Note that in order to do so, we need to add two statements to the initialization part of the literature programming package:

```
\newcounter{imports}\setcounter{imports}{0}\openout 2 \jobname.h
```

It specifies another counter to create a unique number for the imported headers; and opens another output stream for the exported header generation.

In the following we will briefly explain the implementation of the concept import and export when they are encountered in the paper. The structure of the definition of these macros are very similar to those of usage and definitions macros (see lines 6 of the import macro and Line 2 of the export macro. For distinguish them in the generated paper, we use yellow and cyan for the highlight colors.

```
1 \newcommand{\printimp}[1]{\addtocounter{imports}{1}
2 % Open the auxiliary header file
3  \immediate\openout 4 \jobname\arabic{imports}.h
4  \immediate\write 4 {struct #1 { int dummy; }; } \immediate\closeout 4
5  \immediate\write 0 {\string #include "\jobname\arabic{imports}.h" } }
6 \newcommand{\import}[2]{\printimp{#1}{\color{yellow} #2 \color{black}}}
```

The parts for implementation of the C/C++ program generation respectively are shown in lines 1-4 and Line 1 of the two macro definitions.

```
1 \newcommand{\printexp}[1] {\immediate\write 2 {struct #1 {int dummy;};}}
2 \newcommand{\export}[2]{\printexp{#1}{\color{cyan} #2 \color{black}}}
```

The C/C++ statement for importing a concept is actually a struct definition statement, similar to the one output from the definition macro. However, we places the struct definition in a header unit file, other than embed it into the main compilation unit. Meanwhile, the compilation unit has an #include statement generated in place.

```
\#include "<jobname><nImport>.h"
```

where the file `<jobname><nImport>.h` contains a definition of the concept being imported.

The C/C++ statement for exporting a concept is shown similarly. Note that the outputted header for the exported concepts are meant for an interface of the paper to the other papers that may cite it. In other words, the C/C++ does not need to include them to check the integrity of the content of the marked paper. Therefore no `#include` statement is generated for the exported concepts.

Program analysis of the generated code Having the C/C++ code generated from the literature programming macros, several kinds of program analysis can be performed.

The easiest type of analysis is a compilation of the generated code. The syntax errors of the generated program captured by a C/C++ compiler can reflect some structural problems in the paper. For example:

- A variable is used without definition.
- A variable/definition is duplicated.
- In the included file a type A is defined without definition of B.

Tracing back the source of these errors, one can immediately tell what kind of structure error happens.

4 A Case Study

In order to investigate the feasibility of our ideas, we actually marked up a published paper, transformed it into a C++ program, compiled it and analyzed the results. We used the publication of [10] as a basis for our case study. As many other potential papers, this paper on aspect-oriented programming has some characteristics that make it a suitable candidate for our purpose: It gives a brief

overview of aspect-oriented programming at the beginning of the contribution and also refers to and imports some technical concepts throughout the paper. At the same time, it assumes that the reader is familiar with some concepts and uses them without explicitly introducing or referring to them.

4.1 Marking up the Paper

Three principal modes of marking up a paper can be distinguished: (1) mark ups are generated by the author of the paper while writing it in two hours (198 macros were introduced and 5 errors were detected and fixed)[3]; (2) mark ups are generated by a reader / reviewer after the paper has been written and (3) mark ups are generated by a text analysis algorithm. In this case study, we followed mode 2: Generating the markups as readers of a published paper. In order to make the concept structures visible, one of the authors of this contribution marked the paper [10] up with the four distinct tags introduced earlier. The LaTeX commands we used to mark up the paper included \use{FooBar}{foo bar is used}, \define{FooBar}{blabla is defined}, \import{FooBar}{ ... \cite{...}}, and \export{FooBar}{...}. The marked up paper thus can represent a fundament for preliminarily transforming the paper into a C++ program.

In the following we will illustrate some excerpts of the marked up paper in LaTeX notation:

```
A \define{JoinPoint}{join point is a point in program execution exposed by the
semantics of the language to possible modification by aspects}.
```

The example above illustrates how the \define macro is used to mark up the definition of a join point in aspect oriented programming. The term within the first pair of brackets refers to the unique identifier for the concept (JoinPoint), while the term within the second pair refers to the actual text that is being marked up (join point). By separating between the two, our approach is able to deal with synonyms, i.e. the usage of different terms to denote the same concept.

```
For example, to access the return value at a \use{JoinPoint}{join point},
one calls the method \use{GetReturnValue} {getReturnValue} on the variable
\use{ThisJoinPoint}{thisJoinPoint}.
```

The next example illustrates how the \use macro is utilized to mark up the usage of a concept. The sentence in this example uses the concept JoinPoint. By assigning the same unique identifier, our algorithm can relate the *usage* of the concept to its *definition*.

```
In object-oriented systems that support \import{ImplicitInvocation} {implicit
invocation[9]}, there are two ways for an invoker to invoke an invokee:
explicit call or implicit invocation.
```

In this example, we illustrate the usage of the \import macro. Instead of introducing a new definition, the authors of the paper import the concept ImplicitInvocation from another paper (which in this case was: *Garlan, D., and*

[3] E.g. see this article being marked up: http://www.cs.toronto.edu/~yijun/lp-markup.pdf

Notkin, D., Formalizing Design Spaces: Implicit Invocation Mechanisms. VDM '91: Formal Software Development Methods, Oct. 1991.) What can be seen here is that the concept the authors import is already reflected in the title of the reference. With the concepts introduced earlier, this can be regarded as to represent a *shallow* rather than a *deep* citation. We can now move on to the next excerpt:

```
The main contribution of this work is \export{NovelSynthesisOfObjectAspectOriented
ProgrammingLanguageConstructsAndDesignMethods} {a novel synthesis of object-and
aspect-oriented programming language constructs and design methods}, including
\export{EosU}{the Eos-U language}, \export{CompilerForProductionCode}{a compiler
able to handle production code}, and \export{EvidenceForSignificantBenefitsIn
AOPDesign} {evidence that suggests that this synthesis has potentially significant
benefits in aspect-oriented program design}.
```

In the example above, we can see how the `\export` macro is being used to mark up the main contributions of this paper that might be imported by other publications. However, it is interesting to note that, in contrast to C++ import/export relationships, what authors intend to export from their paper does not necessarily reflect what other authors might want to import. Still, we expect this kind of analogy to C++ programs to be of special relevance in cases where for example a glossary of terms exists or is being developed (such as in [12]), and the exports of the paper (the defined concepts) can clearly be identified.

```
\define{EosU}{Eos-U is a classpect-oriented version of \import{Eos}{ Eos [23]}}
```

The final example illustrates a situation where an import of a concept is nested within a definition. In the example above, the concept EosU is defined by importing the concept Eos from another paper, and specializing it as "is a classpect-oriented version of" Eos.

4.2 Transforming the Paper into a C++ Program

The paper was transformed into a C++ program by utilizing the LaTeX macro introduced earlier. The result of the marked up paper can be seen at `http://www.cs.toronto.edu/~yijun/work/lp/p59-rajan-markedup.tex.txt`.

After using the LaTeX command (see below) to typeset it, a C/C++ program is generated automatically.

```
pdflatex p59-rajan-markedup
```

The program contains a single compilation unit 'p59-rajan-markedup.c', a single exported header unit 'p59-rajan-markedup.h' and 17 headers 'p59-rajan-markedup*.h' for importing concepts. Subsequently, we used the GNU CC compiler to compile the generated program. Because the program is not supposed to be executable, here we only parse it to generate an object file.

```
gcc -c p59-rajan-markedup
```

4.3 Results of Compilation and Interpretation

As a result of a compilation with gcc, the output reveals the following errors of our marked-up case paper:

```
p59-rajan-markedup.c:27: redefinition of ‘struct EosU’
p59-rajan-markedup.c:53: redefinition of ‘struct Binding’ ...
In file included from p59-rajan-markedup.c:77:
p59-rajan-markedup12.h:1: redefinition of ‘struct ConceptualIntegrity’
p59-rajan-markedup.c:2: storage size of ‘a2’ isn’t known
p59-rajan-markedup.c:15: storage size of ‘a8’ isn’t known ...
```

We can observe that the compiler throws the following types of errors: (1) redefinition of a concept inside the compilation unit; (2) redefinition of a concept in an included header unit; and (3) unknown storage size for the usage of an undefined concept.

Interpreting these errors in the light of the marked up paper, the first type of error translates into a situation where a **concept is defined multiple times**. The compiler points us to four such situations in our case paper. One of these situations refers to the concept *Classpect* that was defined on three occasions (two re-definitions) in our case paper. Investigating this type of situation in greater detail, we can observe that the paper predominately deals with the concept of classpects, and therefore conclude that the iterative introduction of a comprehensive concept seems to be reasonable from a pedagogical perspective. However, scattering the definition of a concept across too many different locations within a paper can significantly impair the ability of readers to grasp its meaning, and therefore can represent a potential flaw in the structure of a paper. An improvement potential could be to begin the paper with a comprehensive definition of the concept, and then refer back to specific aspects of the definition when necessary.

The second type of error translates into situations where the **same concept is imported at multiple occasions** – potentially from multiple, different sources. The compiler points us to a single such situation in our case paper: *AspectWerkz* is imported two times. The first import focuses on the positive aspects of this concept:

```
\import{AspectWerkz}{AspectWerkz} provides a proven solution to the
problem of AspectJ-like programming in pure Java [...]}
```

whereas the second import emphasis drawbacks:

```
Third, \import{AspectWerkz}{AspectWerkz} lacks static type checking of
advice parameters
```

By having that kind of information marked up in our case paper, it is possible to identify sections that refer to external sources. The context in which a paper is positioning itself can thereby be explored in a structured way. One potential benefit of having such type of mark up available is the possibility of algorithmically discovering the context in which *related work* is referred to based on an analysis of the *import* markups in a paper.

The third type of error translates into a situation where a concept was **used without being defined or imported**. The meaning of such situations can be

two-fold: Either the concept used represents a concept that is well understood by the respective research community (and therefore does not need to be defined or imported) or it does not (and therefore should be defined or imported). In our case paper, the concept *Class* was used without being defined or imported - which seems to be reasonable for a research paper submitted to ICSE 2005 where the audience can be expected to be familiar with the concept of a *Class*. However, some more specific concepts were used without being defined or imported in our case paper, such as the *EquivalentMethodBinding* concept. These concepts would have benefited from being introduced explicitly.

A fourth type of error can be identified when compiling the case paper with `g++` instead of `gcc`. Here, the compiler throws an error when a concept was **used before being defined or imported**. While this might be necessary in some cases (for example, using a word without exactly defining its meaning in the early introduction or abstract of a paper), it might also point to logical flaws where the reader is supposed to understand concepts that are only introduced later on in the paper. Our approach could point authors to such situations and suggest to investigate them carefully.

As a further benefit of our approach, the compiler can create a header file containing all concepts that represent the conceptual outcome of a paper. The following generated header shows the concepts of our case paper that have been marked up with the *export* command:

```
struct Classpect { int dummy; };  /these represent the exports
struct EosU { int dummy; };
struct EaseOfLearningAspectOrientedMethods { int dummy; };
struct EaseOfUseAspectOrientedMethods { int dummy; };
struct NovelSynthesisOfObjectAspectOrientedProgrammingLanguage { int dummy; };
struct EosU { int dummy; };
struct Classpect { int dummy; };
...
```

Having such kind of explicit representation can provide readers with a clear statement regarding the major contributions of a specific paper.

4.4 Analysis and Reflection

We made the following observations about problems during the course of this study. While marking up the paper, defining what constitutes the export of a paper appeared a hard thing to do. This strongly depends on the style of a paper, the domain in which it is placed and the context of available literature. In some papers, the export might refer to a new idea, a developed framework, a critical discussion or other aspects. Identifying export concepts and naming them sensibly represented a major challenge in this case study. Furthermore, identifying what constitutes an import and what constitutes a definition or usage of a concept is not always that easy to distinguish either. To resolve that problem, we agreed that whenever the usage of a concept is accompanied by a literature reference, it represents an import. In some cases, the definition of a concept might also be distributed across the paper (e.g. incremental introduction of the concept

for pedagogical reasons). In such situations, the analogy to C++ programs does not hold well. In this case study, we addressed this issue by marking up the first attempts of defining a concept in the paper.

While we encountered a series of issues during the course of the study, however, we are optimistic by the fact that many of these problems dissolve when a paper is marked up in mode 1 (where mark-ups are generated by authors of the paper rather than readers). This also gives authors the benefit of being able to check the structure and relationships between major concepts of their paper already *before* submitting the paper to a conference or a journal.

5 Discussions

The rules defined in section 2 can be partially enforced by the case study described in section 4, where one paper is converted into a C/C++ program. Namely Rules 1–3 can be directly checked by a C++ parser, whereas Rules 4 and 5 can be enforced with a header restructuring tool when all the cited papers are processed. During the study, we foresee many potential pragmatic uses of a literature programming tool. Among them, it is easy to see four major gains in the application: saving human efforts, providing common exchanging concepts, reusing existing program understanding tools, and reusing established software engineering processes such as agile development.

Human effort It was initially thought a barrier for literature programming to annotate or markup the document manually for the concepts, especially by the readers or reviewers. As we were doing this exercise for a full length 10-page IEEE conference paper, it is found reasonably light-weight in doing such annotations. The number of concept definitions is typically smaller than half the number of paragraphs, the number of concept imports is proportional to the number of citations, and the number of concepts export is even smaller, usually proportional to the number of keywords. The only unpredictable is the number of usages, as it is larger than the total number of definitions and the imported concepts. At this point, we only support a limited number of markup macros, we found it very easy to define editing macros in a text editor (e.g. VIM) to perform such markup tasks using simple keystrokes.

Human effort can be saved greatly when the full text of a document can be mechanically marked up with semantic tags, reported consistently ¿70% precision in recover the concepts [9]. Notably such markup procedure also reused a programming language parser TXL.

Ontology As observed in our experiments, the import/export macros uses or constructs an interface to the papers as a collection of concept definitions. Often a concept is not defined before its use is not considered as an structural error, if the concept being imported is a jargon term commonly accepted by the reader community. However, not every reader would agree on such commonality, even a special interests group of people (e.g. the aspects researchers are a subgroup

of the software engineering community) may agree on different sets of terms being considered "common". Therefore, the same presentation, when facing different audiences, may have to define different sets of concepts as the basics for discussing the other part of the paper. Similarly, if there was an ontology for the particular domain, it would be much easier to simply "import" the domain concepts into the paper without explicitly mentioning them.

In such cases, we can regard a header to be a collection of concept definitions. When the paper of a certain domain is analyzed, the domain-specific concepts will be included implicitly to avoid reporting extraneous irrelevant errors.

Programming tools Besides the parsers, many software analysis tools can be reused to deal with document understanding problems [9]. For example, automated software engineering tools such as class diagrams, software architecture, program dependency analysis, program slicing, program refactoring, program measurements, software clustering etc, can all be extensively used for the sake of literature programming. For example, in answering how many papers cites the paper, one can use the citation graph as a similar construct as useful as a call graph in the traditional programming tool. The reason that programming tools can be reused is very straightforward, i.e., the target subject of literature programming is a program. Getting the right programming tool to do the literature analysis, could greatly increase the productivity of literature analysis. If we already have a header exported from the source of the cited paper, of course, we do not need to produce the header unit, rather just use it. Therefore we provide an alternative implementation of the import, through modifying the implementation of the `\cite` macro. In other words, the cite-graph can be embedded into the finer-grain program unit dependencies that allows us to analyze the false citations and deeper citations problems in the near future.

A software engineering process becomes agile as the team is more flexible and the environment demands unpredicted changes [13]. An automation for agile development is achieved by a continuous integration process [14]. In such a process, changes are monitored such that they are triggers to the build, release and regression testing processes. As a result, we can propose a continuous integration framework for the paper construction process, where authors can ignore the build process for the resulting document (e.g. PDF) so as to focus on more creative writings. During the build process, literature programming tools can be used together with the traditional spelling and grammar checking tools to point out the potential structural errors in the paper as warning messages.

6 Related work

Generating paper using a program The following news hits the headline last year that three MIT thesis students have created a program which generates a technical paper accepted by a technical international conference [15]. The purpose of our work is the opposite, that is, to convert a paper written by human authors into a program such that program analysis tools can be applied to detect flaws

in the original paper. And if possible, to restructure that paper using program restructuring tools (e.g. [2]). Such effort can be regarded as the return of the spirit of literate programming: associates programs with documents that can annotate each other [3].

Literate programming D. Knuth proposed *literate programming* as technical solutions that unify document and code into a single source [3]. Applying the literate programming tool such as WEB [3], CWEB [4], NOWEB [5], etc., the source can be split into a document with exact code snippets and a program with well-documented comments. Unlike the the intention of that work, this paper reuses existing programming tools on the literature by converting a traditional paper (document) into a structurally equivalent program. The resulting program is not executable, however, analyzable subject to reusable software engineering and programming tools.

Digital libraries: ACM portal, IEEE explore, Citeseer, DBLP, CiteULike, Springer, Amazon etc. These organizations have classified most of the published computer science literature in bibliographic format in one-way-or-another. These preclassified digital libraries not only point out how many times a paper is cited, but also the citation depedencies between papers. More precisely as in Citeseer, such dependencies can be analyzed into a citation graph [16], of which the structure was analyzed on basis of the citation relations among the literature in the domain of Computer Science (Citeseer). Finding in [16] is interesting as it shows the topological regularity in the dependencies among the papers. In our proposal, such dependencies happen at finer granularity involving concepts inside the papers. The finer-grain literature programming analysis is enabled only when one can convert them into program units.

Header restructuring. Header restructuring is a precompilation tool that cleans up the header units of the software and reorganizes them into clean relations without *redundancies* (two duplication definitions) and *false dependencies* (one change in an element can incur an unexpected change on the elements that does not have to change) [2]. Evidence has shown that applying such header restructuring, a complex program can be compiled in less than half of the time because the expand form has half of the original size. After the papers in literature are converted into C/C++ programs with inclusion dependencies recording the citation relations, they are subjected to similar restructurings: redundant definitions and false citations can be removed as removal of redundancy and false dependencies, deep citations can be shortened by citing the paper that do contain the definition of the concept. These restructurings, depending on how severe the structural problem is, can help shorten the paper or reduce the time it takes to find the definitions of all the concepts. No all concepts need to be defined in one paper, especially those that are familiar to the readers. Nevertheless, given a "standard" pool of concepts, one can prepare a common header to avoid really defining them whereas the structure for introducing other unfamiliar concepts can still be improved by such restructuring.

7 Conclusion

In order to reuse existing program analysis tools for detecting some writing errors in technical papers, we sketched a methodology to convert a technical paper into a program. To demonstrate, we implemented a LaTeX package that is able to convert a paper in LaTeX source form into a C/C++ program. We applied the newly defined macros to a published paper in the prestigious software engineering conference as a case study. The generated program is parsed by the GNU C/C++ parser to detect a number of parsing errors. We analyzed these errors and found out some of them are due to the experimental usage of markups, whereas others are revealing presentation issues that could have been addressed better. Furthermore, we explained the possible applications of literature programming to analyze the literature by reusing other programming tools such as header restructuring, transformation systems, program dependency visualizations, etc. This work as reported so far has not completed the experiment to show the full potential of literature programming, as our experiment only focused on marking up a single paper. Given that we have implemented the easy-to-use literature programming package, when marking up multiple papers, we expect to see more potentials and problems, since there would be a need for managing the dependency relationships between them.

References

1. Turing, A.: Computing machinery and intelligence. Mind **59** (1950) 433–460
2. Yu, Y., Dayani-Fard, H., Mylopoulos, M., Andritsos, P.: Reducing build time through precompilations for evolving large software. In: International Conference on Software Maintenance (ICSM). (2005) 59–68
3. Knuth, D.: Literate programming. Comput. J. **27(2)** (1984) 97–111
4. Knuth, D., Levy, S.: The CWEB System of Structured Documentation: Version 3.0. Addison-Wesley Longman Publishing Co., Inc. (1994)
5. Johnson, A., Johnson, B.: Literate programming using noweb. Linux J. (1997) 64–69
6. Li, V.O.K.: Hints on writing technical papers and making presentations. IEEE Transactions on Education **42(2)** (1999) 134–137
7. Gamma, E., et al.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (1995)
8. Yu, Y., Lu, J., Mylopoulos, J., Sun, W., Xue, J.H., D'Hollander, E.H.: Making XML document markup international. Software: Practice and Experience (**35(1)**) 1–14
9. Kiyavitskaya, N., Zeni, N., Cordy, J.R., Mich, L., Mylopoulos, J.: Applying software analysis technology to lightweight semantic markup of document text. In: ICAPR'05. (2005) 590–600
10. Rajan, H., Sullivan, K.: Classpects: unifying aspect- and object-oriented language design. In: Proceedings of International Conference on Software Engineering (ICSE). (2005) 59–68
11. Eijkhout, V.: TeX by Topic, Chapter 30. Addison Wesley (1992) 307.
12. Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., Sabetzadeh, M.: A manifesto for model merging. In: Proceedings of the 2006 international workshop on Global integrated model management, ACM Press (2006) 5–12
13. Cockburn, A.: Agile Software Development. Addison Wesley (2002) 256.
14. Fowler, M., Foemmel, M.: Continuous integration, fetched from http://www.martinfowler.com/articles/continuousintegration.html. (2005)
15. Stribling, J., K.M., Aguayo, D.: Scigen - an automatic cs paper generator. Technical report, MIT (2004)
16. An, Y., Janssen, J., Milios, E.E.: Characterizing and mining the citation graph of the computer science literature. Knowl. Inf. Syst. **6** (2004) 664–678