International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 6, Number 2, pages 263–289, July 2016

Xevtgen: Fortran code transformer generator for high performance scientific codes

Reiji Suda
Graduate School of Information Science and Technology, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113–8656, JAPAN

Hiroyuki Takizawa
Graduate School of Information Sciences, Tohoku University
6-6-01 Aramaki-aza-aoba, Aoba-ku, Sendai, 980–8579, JAPAN

Shoichi Hirasawa
Graduate School of Information Sciences, Tohoku University
6-6-01 Aramaki-aza-aoba, Aoba-ku, Sendai, 980–8579, JAPAN

Received: February 15, 2016
Revised: May 6, 2016
Accepted: July 8, 2016
Communicated by Kazuhiko Komatsu

Abstract

High performance scientific codes are written to achieve high performance on a modern HPC (High Performance Computing) platform, and are less readable and less manageable because of complex hand optimization which is often platform-dependent. We are developing a toolset to mitigate that maintainability problem by user-defined easy-to-use code transformation: The science code is written in a simpler form, and coding techniques for high performance are introduced by code transformations. In this paper, we present `xevtgen`, which is a code transformer generator of our toolset. Transformation rules are defined using dummy Fortran codes with some directives, and we expect our design makes our tool easier to learn for Fortran programmers. Some examples of code transformations, as well as an application to a real scientific application, are shown to discuss the practicality of the proposed approach. `Xevtgen` assumes XSLT as a backend, and generates an XSLT template from the dummy Fortran code. That design of `xevtgen` exploits the power of XSLT, and inherits some limitations of XSLT. In our plan, those limitations will be mitigated by additional tools in our toolset.

Keywords: Fortran, Code transformation, Code generation

1 Introduction

As clusters and supercomputers are widely available nowadays, high performance scientific codes tend to have tens or hundreds thousands of lines of codes. There may be two major reasons of those large sizes of codes. One reason is that, as the processors' clock frequency almost no longer increases, we have to exploit advanced features of latest computer architecture: parallelism, memory hierarchy, etc. That requires more and more complex coding, deeply dependent on the computer architecture of a specific execution platform. The other reason is that, because higher performance is provided by advanced hardware, scientists choose more complex simulation models for their analysis, which were not possible in the past. Those two reasons will continue to exist, perhaps more intensively, in the future.

In this work, we are concerned with the former reason of the complexity of high performance scientific codes. The platform-dependent coding makes the codes less readable and less manageable, and harder to port to a new platform. Our project aims to mitigate that coding complexity in high performance scientific codes, by using *code transformations*. One of challenges we are tackling in our project is to design an appropriate interface of such code transformation rules that is easy for science programmers, who are familiar to only one or very few programming languages. Compiler experts could develop a custom compiler or a code transformer for their own code transformations, but it is not so easy for average programmers to develop such a tool by themselves. In this paper, we present `xevtgen`, which is a tool that generates code transformations. An advantage of `xevtgen` is that the code transformation rules can be described in conventional Fortran plus some special directives, without much knowledge on theory and practices about compilers. Therefore, Fortran programmers can learn and use our tools more easily than developing a custom compiler or a code transformer.

`Xevtgen` is based on XSLT, which is one of the most well-known and well-available XML transformation framework. This paper also reports how `xevtgen` generate an XSLT template from a dummy Fortran program. By using XSLT as transformation engine, we can exploit the power of XSLT in our toolset. But this design also inherits some limitations of XSLT. We explain our plan in which those limitations will be mitigated by additional tools.

This paper consists of 10 sections. Section 1, this section, is an introduction. In Section 2, an overview of our approach and toolset is explained. From Section 3, the main part of this paper begins. Section 3 is a brief introduction of how to describe code transformations in `xevtgen`. Section 4 explains simple and straightforward examples of code transformations. Section 5 explains a little more elaborated examples from our development of autotuning software using `xevtgen`. Section 6 explains an application of `xevtgen` to a real-world application program for performance portability. In Section 7, the implementation of `xevtgen`, that is, how Fortran dummy codes are translated into XSLT templates, is explained. Section 8 describes some limitations of `xevtgen` by itself, and reveals our plan to solve them. Section 9 compares our work to related work. Section 10 gives concluding remarks.

2 Overview

2.1 `Xevolver` approach

It is frequently said that "coding what to compute" and "coding how to compute" are not separated well in lower-level programming languages, such as Fortran and C. By using higher-level languages such as MATLAB, or domain specific languages (DSL) such as stencil programming DSLs[3], programmer can concentrate on "what to do." However, regrettably, some languages cannot utilize full power of the newest computers, and some languages cannot provide enough programmability for general purpose computing. Therefore, many HPC scientific codes are still written in Fortran, C and C++.

Another reason to use conventional programming languages is that there are many existing codes, developed and maintained for decades. Therefore, in our project, we assume such existing legacy codes, and develop a methodology and a toolset to achieve high performance in such codes without introducing major modifications of the existing codes.

Our aim is to separate "how to compute" in such codes, especially on high performance computers, from "what to compute." We chose to use *code transformation* for that purpose[16]. That is, the source code is mostly written to describe "what to compute," and there are separate code transformation rules that modify the source code toward high-performance, perhaps platform-dependent codes. The high performance computer architecture in an extreme-scale computing era will be different from that in the present, but using our toolset, the original code itself does not need to adapt to the new architecture if separate code transformation rules are properly written for the new architecture.

The platform-dependent optimizations should be generally done by compilers developed for those platforms. In reality, it is hard to write an optimized code which outperforms very-well-developed optimizing compilers. Still, there are many occasions where compiler optimizations are not applied as the code developers expect. Compilers are conservative, that is, they do not apply optimization unless they are sure that the behavior of the program does not change. Some other reasons are that, for example, the code is too long to analyze, some analyses such as aliasing cannot be fully solved, or compilers cannot prove safety of some optimizations, which is known to be safe by the developers. We observe that, there are always gaps between compiler optimizations and developers' expectations.

Based on that observation, we are developing `Xevolver`, a code transformation framework, and its utility toolset `Xevolver tools`. We refer to the whole set of software just as "`Xevolver`." `Xevolver` is expected to fill the gap between compiler optimizations and developers' knowledge about possible optimizations. In this paper, we assume that the developers know the gap, and know how to modify their codes for higher performance. Our aim is to provide code transformation tools for code developers. Although we are also planning to support C and C++, this paper presents only the Fortran version which is under active development.

2.2 Xevolver Tools

In this paper, we introduce `xevtgen`, which is a part of `Xevolver tools`. `Xevolver tools` consist of four parts. The first part is `xevparse/xevunparse`, which parses a Fortran source code into an XML document using `Xevolver`. The Fortran source code is translated into an XML tree which represents an Abstract Syntax Tree (AST) of the source code. The XML format is defined by `Xevolver`. The second part is `xevtgen`, which is the main topic of this paper. The input of `xevtgen` is a Fortran-like code which describes a set of code transformation rules, and its output is a template file of XSLT. XSLT is one of the standard specifications to describe XML data conversion rules. Thus, by using `xevtgen` and an XSLT engine, we can transform an XML document representing the original Fortran code so that the transformed XML document accordingly represents another Fortran code. `Xevunparse` can back-translate the transformed XML document to its corresponding Fortran code. The other two parts are `xevdriver` and `xevutils`. The `xevdriver` provides a simple script language to control code transformations using our toolset. The `xevutils` provide some miscellaneous functions that are useful in transformations, for example, choosing new names, controlling the order of code transformations, helping debugging of code transformations, modularize set of code transformations, and so forth.

3 Xevtgen

We develop `xevtgen`, because most Fortran programmers are unfamiliar to XSLT and similar XML transformation systems, even to XML. For such Fortran code developers, it is really painful to learn XML and XSLT. Our experience is that XSLT is quite hard to debug, unless the developer is very well familiar to most features of the XSLT coding. This is perhaps because XSLT assumes a little simpler XML transformations than that of a practical Fortran AST. Motivated by this, we designed `xevtgen` so that the user does not need to know XML and XSLT at all.

Before explaining definitions of code transformations, we explain command line usage of `xevtgen`. The `xevtgen` command takes two arguments:

```
    xevtgen infile outfile
```

Here the first argument *infile* is the file with code transformation definitions, and the second argument *outfile* is the name of the file to which XSLT translation rules are output. The *infile* is formatted as a Fortran program, as is discussed below, and the *outfile* is in XML. (It is more precise to say that *infile* is an XML document which is converted from a Fortran-like code by `xevparse`. However for simplicity, we call the input file of `xevparse` as *infile* of `xevtgen`.)

To transform a program `src.f90` to `dst.f90` by using the XSLT code transformation rules generated as *outfile*, one should run the commands as follows:

```
xevparse src.f90 src.xml
xsltproc outfile src.xml > dst.xml
xevunparse dst.xml dst.f90
```

`Xevolver` is developed based on ROSE compiler infrastructure[12], and hence supports most features of Fortran 77/90/95 and 2003.

3.1 Infile format

The *infile* describes the code transformation. Its basic format is just a dummy Fortran program — here a "dummy" program may do no meaningful computation, but must be conformant with the Fortran programming language — plus some directives. The generic directive format is as follows:

$$\begin{aligned} \textit{directive} &:= \texttt{!\$xev}\ \textit{clause}\ [\textit{clause}\ldots] \\ \textit{clause} &:= \textit{name} \mid \textit{name}\texttt{(}\textit{term}[\texttt{,}\textit{term}\texttt{,}\ldots]\texttt{)} \\ \textit{term} &:= \textit{name} \mid \textit{int} \mid \textit{str} \mid \text{`}\textit{exp}\text{`} \\ &\quad \mid \textit{name}\texttt{(}[\textit{term}[\texttt{,}\textit{term}\texttt{,}\ldots]\texttt{)} \end{aligned}$$

where *int* is an integer, *str* is a string, and *exp* is a Fortran expression. Integers and strings are in the Fortran format. A *name* consists of alphanumerics, underscores, and periods. Each directive starts with `!$xev`, followed by one or more clauses. Each clause has a name, possibly followed by arguments, i.e., a list of terms enclosed by parentheses. Each term is one of name, integer, string, Fortran expression, or name with arguments. A directive with a `begin` clause corresponds to a directive with an `end` clause, and they form a single *statement*. A directive without `begin` is a single statement by itself.

3.2 Defining a code transformation in `xevtgen`

An `xevtgen` *infile* defines one or more code transformations. The most basic transformation is literal replacement. For example, if *infile* contains a directive

```
!$xev tgen trans exp src(‘N‘) dst(‘10**3‘)
```

then `xevtgen` generates an XSLT transformation rule that converts all occurrences of variable `N` into integer `10**3`. Here the first two clauses `!$xev tgen` represent that the directive is of `xevtgen` rule. The next clause `trans` introduces a transformation rule. The following clause specifies the type (explained below) of the transformation rule. In this example, `exp` implies that the rule is a transformation from an expression to another expression. The expression in the `src` clause, which is `N` in this example, defines what pattern of Fortran code fragment is extracted to be transformed. We call it *source pattern.* Expression in the `dst` clause, which is `10**3` in this example, defines the result of the transformation. We call it *destination pattern.*

There are six types of entities in `xevtgen`:

- *Name*: names of variables, functions, modules, named labels of DO and IF statements, etc. in Fortran, and name of clauses and terms in directives.
- *Value*: integer or string values, either in Fortran program or in directive.
- *Clause*: clause of directive.
- *Term*: term of directive.
- *Exp*: Fortran expression.
- *Stmt*: statement of either Fortran program or directive.

Transformation rules of terms and expressions are defined as in the above example. Transformation rules of clauses and statements are specified with two directives, for example:

```
!$xev tgen trans stmt src begin
    IF (I .EQ. 0) EXIT
!$xev tgen trans stmt src end
!$xev tgen trans stmt dst begin
    IF (I == 0) THEN
        EXIT
    END IF
!$xev tgen trans stmt dst end
```

Currently, `xevtgen` does not support transformation of names and values. One may want to change the name of a function from `foo` to `bar`. Transformation of names will provide a one-step solution to this requirement. But because of some technical issues caused by the XML structure and limitations of XSLT, such a solution is not provided in `xevtgen`. We are planning to provide a separate tool to change names and values.

3.3 Tgen-variable

Code transformations need *meta-variables*, which enable transformations with some parameters. In `xevtgen`, they are called *tgen-variables*. Tgen-variables work in two ways: one is a wildcard in the source pattern, and the other is to convey information from a source pattern to its destination pattern. The following is a simple example of using a tgen-variable.

```
!$xev tgen var(a) exp
!$xev tgen trans exp src(‘sq(a)‘) dst(‘a*a‘)
```

In the first line, `a` is declared as a tgen-variable of type *exp*. In the source pattern `sq(a)`, it matches any call of a function named `sq` with a single argument, since `a` behaves as a wildcard. In the destination pattern `a*a`, the subexpression `a` of the source pattern `sq(a)`, that is, the argument of `sq`, is copied into each occurrence of `a`. Thus,

```
k = sq(4) + sq(b) + sq(3+c)
```

is transformed into

```
k = 4*4 + b*b + (3+c)*(3+c)
```

Here, a small difference from macros of C language is that, the parentheses of `(3+c)*(3+c)` automatically appear, since the replacement is done at an AST level, not at a text level. If the tgen-variable appears in the source pattern, then the transformation rules are applied to the tgen-variable. For example, `sq(sq(3))` will be transformed into `(3*3)*(3*3)`. But the recursive application can be prohibited by adding `norec` clause in the tgen-variable declaration.

In many cases, tgen-variables appear as named variables in the pattern, as in the example above. There are some special forms. The following example contains uses of tgen-variables of type *stmt*.

```
!$xev tgen list(body_if, body_else) stmt
!$xev tgen src begin
    IF (.false.)  THEN
        !$xev tgen stmt(body_if)
    ELSE
        !$xev tgen stmt(body_else)
    ENDIF
!$xev tgen src end
!$xev tgen dst begin
    !$xev tgen stmt(body_else)
!$xev tgen dst end
```

Here, `!$xev tgen stmt(`name`)` behaves as one single tgen-variable of type *stmt*. One cannot write just `body_if` instead of `!$xev tgen stmt(body_if)`. This is because we use dummy Fortran code to represent transformation. As `body_if` is not a valid Fortran *statement*, it is not accepted by the parser.

This example introduces two more things: First, `trans stmt` can be omitted in `!$xev tgen src` and `!$xev tgen dst`, since they are so frequently used. Second, `!$xev tgen list(`name list`)` declares tgen-variables of *list kind*, matching lists of non-negative numbers of entities.

In the above example, `!$xev tgen stmt(body_if)` catches all statements between `IF` and `ELSE`, and `!$xev tgen stmt(body_else)` catches all statements between `ELSE` and `ENDIF`. The present code transformation conducts, if the condition of `IF` is a constant `.false.`, then the IF sentences is removed, and only the body of ELSE-part remains.

Note that the above transformation is not safe in general. The body of THEN-part may contains a statement with a label, and there may be a GOTO statement with that label. At least at the current status, our toolset does not provide a checking mechanism of such an unsafe transformation. It should be emphasized that our purpose is to provide code transformations to avoid modifying the original code, and thus, as is the case in conventional manual code modification, programmers still need to be responsible for the correctness or safety of their code transformations.

Figure 1 shows a little long example, which splits a loop into two. A source code is assumed to have a loop, for example,

```
!$xev loop split
    DO k = 1, N
        a(k) = ...
        !$xev split point
        b(k) = ...
    END DO
```

and after transformation, it will become

```
    DO k = 1, N
        a(k) = ...
    END DO
    DO k = 1, N
        b(k) = ...
    END DO
```

that is, there become two loops, keeping loop indices and ranges, and the statements before `!$xev split point` are stored in the body of the first loop, and the statements after `!$xev split point` are in the body of the second loop. This *loop splitting*, also known as *loop fission*, sometimes improves locality of computations, and is one of the basic loop transformation techniques for HPC. Generally, a scientific code needs to be modified for achieving high performance because the compiler does not necessarily work as expected. Therefore, this kind of basic loop transformation techniques are yet needed for performance tuning of practical scientific codes.

The transformation rule contains three definitions of transformations. The first transformation (lines 5 to 19) is an initialization. It finds a DO loop preceded by `!$xev loop split`. Then it creates the first loop with an empty body, and with the same index variable and range. Here, the increment `i2` is included in the pattern, but it matches with DO loops without increment. This is a little tricky, but convenient to write slightly general transformation rules. Next see the third transformation (lines 41 to 62), which moves the first statement in the second loop to the last statement in the first loop. Last, the second transformation (lines 21 to 39) is a finalization. If the first statement of the second loop is `!$xev split point`, then the split has been successfully done. It just removes some directives.

Only one of them is applied at once. The transformation is assumed to be applied repeatedly, until no transformation is done any more. The order of three transformation rules is significant. If the code matches multiple transformations, then the transformations defined first is applied. This prioritization on earlier transformations is different from XSLT, but is conventional in programming languages with pattern matching functionalities.

3.4 Conditional transformation

Users may need to choose transformations depending on some conditions. If the condition appears in a matching pattern, then just defining multiple transformations may be enough, placing a more specific matching pattern before a more general matching pattern.

Another possible condition is related to an ancestor node in an AST. For example, one statement should be transformed only if it is within the definition of a specific function. In `xevtgen`, such a condition can be specified by a *context*. Such a context can be defined as

```
!$xev tgen ctxdef(infoo) stmt begin
    FUNCTION foo(args)
        !$xev tgen stmt(body)
    END FUNCTION
!$xev tgen ctxdef end
```

and is named as `infoo`. The defined context is referred in the matching pattern as

```
!$xev tgen src context(infoo) begin
    ...
!$xev tgen src end
```

Then the transformation is only applied within the definition of the function `foo`.

Anything which can be specified as a source pattern can be a context. The pattern should contain one or more tgen-variable(s). When the transformation rules are applied to a code fragment which corresponds to such a tgen-variable, the context is regarded as valid, and the rules with the context are applied. Otherwise, the context is regarded as invalid, and the rules with the context are ignored. In the above example, `!$xev tgen stmt(body)` is the tgen-variable in the context definition, and it corresponds to the body of the function definition. Thus the transformation rule with `context(infoo)` is applied only to the code fragments within the definition of function `foo`. By using contexts, one can define, for example, rules which are valid only within some specified DO loop, within an argument of some function call, or within some specified directive structure.

Some additional functionalities related to contexts are provided by `xevtgen`. One is logical operations of contexts, such as `context(and(ctx0, or(ctx1, not(ctx2))))`, where `ctx0`, `ctx1` and `ctx2` are context names. Another feature is a context for another context: If a context (say, `ctx1`) definition is conditioned by another context (say, `ctx0`), then the context `ctx1` will be active only when it is in context `ctx0`. A context (say, `ctx2`) can cancel another context (say, `ctx1`). Those features are designed to resemble scoping rules in many programming languages.

The third method of limiting matching patterns is *conditions*, which is designed to catch more detailed conditions in the matching pattern. A condition is defined by `condef`, for example, as follows:

```
!$xev tgen var(x) exp
!$xev tgen condef(upd) stmt contains begin
    i = x
!$xev tgen condef end
```

The condition `upd` becomes active if the variable `i` is updated by an assignment (note that `x` is a tgen-variable, so can be any expression). Then it can be used, for example, as follows:

```
!$xev tgen var(i0, i1, i2) exp
!$xev tgen list(body) stmt condition(upd)
!$xev tgen src begin
    DO i = i0, i1, i2
        $xev tgen stmt(body)
    END DO
!$xev tgen src end
```

Here, the condition is referred to as `condition(upd)` in the declaration of tgen-variable `body`. Then, if a source pattern contains a reference to the tgen-variable `body`, then it fires only when the code fragment that corresponds to `body` satisfies the condition: in this case, contains an assignment statement to the variable `i`. So the above pattern matches if the loop index `i` is updated within the loop body, and perhaps one can warn the programmer. The assignment to `i` can be found within a deeper program structure, for example, within an inner DO loop or an IF construct within the outer DO loop indexed with `i`.

There are four kinds of conditions in `xevtgen`: `is`, `same`, `contains`, and `listwith`. The condition `is` means that the tgen-variable directly matches the pattern given by the condition. It is useful when the condition definition contains tgen-variables, and when the condition is used in logical operations `and`, `or`, and `not`. The condition `same` means that the tgen-variable has the same name or value as the condition specifies, which can be used only for names and values. Condition `same` for other types, such as expression and statements, are not supplied in `xevtgen`. Such a condition corresponds to a unification, and XSLT does not provide unification, so `xevtgen` does not. Section 5.1 introduces a technique which is similar to unification. The condition `contains` means that the tgen-variable contains the specified pattern. The condition `listwith` is activated only when the specified pattern is an element of the list that the tgen-variable represents.

As described above, `xevtgen` allows users to define transformation rules without any knowledge about compilers, XML, nor XSLT. Basically, users just need to write two versions of a code, the source pattern and the destination pattern. Moreover, combining matching patterns with contexts and conditions, `xevtgen` can provide many matching conditions which is available in XSLT. But the possible set of transformations of `xevtgen` is less than that of XSLT. For example, one can write XSLT template with character string manipulations, which is not made available in `xevtgen`. In a sense, that restriction reduces the risk of defining syntactically illegal rules.

Overall, `xevtgen` has been designed and developed so that it can be used to define code transformations frequently required in performance tuning of scientific codes. The following sections will show several use cases to demonstrate that `xevtgen` can define various code transformations useful in practical programming.

4 Simple examples

This section shows two use cases to exemplify the expression ability and limitations of `xevtgen` (rather than performance tuning with `xevtgen`).

4.1 Choose

The first example is an implementation of *choose*, which is something like the conditional operators `p?x:y` in the C language. One can use `choose(p, x, y)` in expressions, which is expanded to IF sentences. Note that `choose(p, x, y)` cannot be implemented as a function, because `y` should be evaluated only if `p` is true, and `y` should be evaluated only if `p` is false.

In this paper, we focus on a very simple implementation, to keep the explanation brief. First, we assume there is only one occurrence of *choose* in a program. If there are many *choose* instances, they must be translated one by one, and such a control will be achieved by using `xevutils` and `xevdriver` of our toolset. Second, we assume a *choose* expression in an assignment, which is simplest to treat. If *choose* appears in the condition of an IF statement, it would be better to evaluate the condition before the IF statement.

It is possible to define *choose* in one file, but for simplicity, we explain it in three transformations. The first transformation is to find *choose* and make an IF statement.

```
!$xev tgen var(p, x, y, u) exp
!$xev tgen condef(has_choose) contains exp(‘choose(p,x,y)‘)
!$xev tgen var(chexp) exp cond(has_choose)
!$xev tgen src begin
    u = chexp
```

```
!$xev tgen src end
!$xev tgen dst begin
    IF (p) THEN
        !$xev choose_then
        u = chexp
    ELSE
        !$xev choose_else
        u = chexp
    END IF
!$xev tgen dst end
```

An assignment of some expression *chexp* that contains *choose* is transformed to an IF statement with condition *p*, the first argument of *choose*. The assignment statement is copied to both THEN part and ELSE part, preceded by directives with `choose_then` and `choose_else`, respectively.

The second transformation is as follows. If an expression `choose(p, x, y)` is in a statement followed by `choose_then`, then it is transformed into just `x`. If it is followed by `choose_else`, then it is transformed into `y`.

```
!$xev tgen var(s) stmt
!$xev tgen var(p, x, y) exp

!$xev tgen ctxdef(ch1) stmt begin
    !$xev choose_then
    !$xev tgen stmt(s)
!$xev tgen ctxdef end
!$xev tgen trans exp src(`choose(p,x,y)`) dst(`x`) context(ch1)

!$xev tgen ctxdef(ch2) stmt begin
    !$xev choose_else
    !$xev tgen stmt(s)
!$xev tgen ctxdef end
!$xev tgen trans exp src(`choose(p,x,y)`) dst(`y`) context(ch2)
```

The third transformation is just to remove `choose_then` and `choose_else`, which is not shown here, because it is trivial. Let us apply these transformations to the following Fortran code:

```
function test(a, b)
    integer ::  a, b, test
    test = choose(a < b, -1, 1)
end function
```

After the first transformation, it will become:

```
function test(a, b)
    integer ::  a, b, test
    IF (a < b) THEN
        !$xev choose_then
        test = choose(a < b, -1, 1)
    ELSE
        !$xev choose_else
        test = choose(a < b, -1, 1)
    END IF
end function
```

Then it is converted by the second transformation:

```
function test(a, b)
    integer ::  a, b, test
    IF (a < b) THEN
        !$xev choose_then
        test = -1
    ELSE
        !$xev choose_else
        test = 1
    END IF
end function
```

Finally it becomes as follows:

```
function test(a, b)
    integer ::  a, b, test
    IF (a < b) THEN
        test = -1
    ELSE
        test = 1
    END IF
end function
```

4.2 Unswitching

Unswitching is a widely-used coding technique, that moves IF switches with loop-invariant conditions out of DO loops. Let us first see an example of target code:

```
subroutine test(n, a, ch)
    integer ::  n, ch
    real ::  a(n)
    !$xev loop unswitch
    do i = 1, n
        if (ch .eq.  0) then
            a(i) = 1.0 / i
        else if (ch .eq.  1) then
            a(i) = i
        else if (ch .eq.  2) then
            a(i) = 0.0
        else
            a(i) = 1.0
        end if
    end do
end subroutine test
```

Unswitching translates the code above into the following code:

```
subroutine test(n, a, ch)
    integer ::  n, ch
    real ::  a(n)
    if (ch .eq.  0) then
        do i = 1, n
            a(i) = 1.0 / i
        end do
    else if (ch .eq.  1) then
        do i = 1, n
            a(i) = i
        end do
    else if (ch .eq.  2) then
        do i = 1, n
            a(i) = 0.0
        end do
    else
        do i = 1, n
            a(i) = 1.0
        end do
    end if
end subroutine test
```

The former code is more readable, but the latter code performs better on many platforms. Some compilers can do unswitching automatically, but sometimes cannot, because it is hard for compilers to check whether the conditions are loop-invariant or not.

The following code is a definition of unswitching transformation.

```
!$xev tgen var(i, i0, i1, p, q) exp
!$xev tgen list(body1, body2, body3) stmt

!$xev tgen src begin
!$xev loop unswitch
   do i = i0, i1
      if (p) then
         body1 = xevtgen_var
      else if (q) then
         body2 = xevtgen_var
      else
         body3 = xevtgen_var
      end if
   end do
!$xev tgen src end
!$xev tgen dst begin
   if (p) then
      do i = i0, i1
         body1 = xevtgen_var
      end do
   else
!$xev loop unswitch
      do i = i0, i1
         if (q) then
            body2 = xevtgen_var
         else
            body3 = xevtgen_var
         end if
      end do
   end if
!$xev tgen dst end

!$xev tgen src begin
!$xev loop unswitch
   do i = i0, i1
      if (p) then
         body1 = xevtgen_var
      else
         body2 = xevtgen_var
      end if
   end do
!$xev tgen src end
!$xev tgen dst begin
   if (p) then
      do i = i0, i1
         body1 = xevtgen_var
      end do
   else
      do i = i0, i1
         body2 = xevtgen_var
      end do
   end if
!$xev tgen dst end
```

It consists of two rules, one for three or more conditional branches, the other for two (IF-THEN-ELSE). Our parser converts a series of ELSE IF switches into a nested IF-THEN-ELSE syntax, so the above two cases are enough for any number of IF switches. Some statement like `body1=xevtgen_var` is equivalent to `!$xev tgen stmt(body1)`, which are sometimes slightly more readable.

The above transformation produces a code which is little uglier than expected as:

```
if (ch .eq.  0) then
    ...
else
    if (ch .eq.  1) then
        ...
    else
        if (ch .eq.  2) then
            ...
        else
            ...
        end if
    end if
end if
```

but it is possible to define a code transformation to convert the code above into a simpler one.

5 Some use cases in code generation

`Xevolver` and `xevtgen` are not restricted to transformations that preserve the semantics of the codes. In this section we introduce a use case of `xevtgen` for a kind of meta-programming. We are developing a code generator for mathematical routines of autotuning based on Bayesian statistical modeling[15]. Our generator takes an input file that describes a Bayesian model in a format of Fortran code. Then it outputs an executable Fortran code that computes a fitting of observed data into the given Bayesian model. We do not explain the details, but introduce some transformations that are used in our code generation.

5.1 Triplet transformation

Before explaining the code transformations, we introduce an extension of transformations in `xevtgen`, which we call *triplet transformation.*

The need of triplet transformation comes from the lack of unification in XSLT. Without unification, the matching rule (the source pattern) cannot depend on the source code. For example, we have no way to find the specification of a variable found in the source code, or to find the occurrences of an expression found in the source code. Triplet transformation is a method to mitigate that restriction.

A simple example is a macro. Here we have the following source code:

```
!$xev macro(N, 100)
x += N * N
```

and want to transform it to the code below.

```
x += 100 * 100
```

To do that, we just need an `xevtgen` infile as

```
    !$xev tgen trans exp src(‘N‘) dst(‘100‘)
```

To attain the usual semantics of `macro` directive, expressions `N` and `100` should be specified within the `macro` directive, which is in the source code. But in `xevtgen` the replacement patters must be specified in the *infile* of the `xevtgen`, rather than the code to be transformed.

Triplet transformation is a simple trick to enable it. It consists of three transformations. First, we prepare a *dummy infile*:

```
    !$xev tgen trans exp src(‘aaa‘) dst(‘bbb‘)
```

where `aaa` and `bbb` are dummy expressions. Next, we define another transformation, which we call *specializing rule*, as follows.

```
    !$xev tgen var(xxx, yyy) exp
    !$xev tgen src begin
        !$xev macro(‘xxx‘, ‘yyy‘)
    !$xev tgen src end
    !$xev tgen dst begin
        !$xev tgen trans exp src(‘aaa‘) dst(‘xxx‘)
        !$xev tgen trans exp src(‘bbb‘) dst(‘yyy‘)
    !$xev tgen dst end
```

Those two transformation rules are sufficient to a triplet transformation. At the first transformation, the specializing rule is applied to the *source code* that have one or more `!$xev macro` lines. The source code will be transformed into the following:

```
    ...
    !$xev tgen trans exp src(‘aaa‘) dst(‘N‘)
    !$xev tgen trans exp src(‘bbb‘) dst(‘100‘)
    ...
```

The above transformed code is used at the second transformation, and is applied to the dummy infile. It results in the following code, which we call *specialized infile*:

```
    !$xev tgen trans exp src(‘N‘) dst(‘100‘)
```

which is exactly what we wanted to have. At the third transformation, the specialized infile is applied to the source code, and the macro transformation is done. (In this example, the source code can have only one macro definition. To treat multiple macro definitions, we need an external helper that sequentialize multiple transformations.)

A triplet transformation is illustrated in Figure 2. Here, the thick boxes represent the files exist before the transformations and the thin boxes represent the files created by the transformations. The thick arrows represent sources and destinations of the transformations, and the thin arrows show transformation rules. It looks much more complex than a simple transformation, but the only added file is the specializing rule, which inserts some information of the source code into the transformation rule.

5.2 Finding array size

Next, a few transformations used in our code generator are explained. The first transformation finds the array size specification, which is given as follows.

```
real :: xxx(100)

!$xev tgen list(aaa) exp
!$xev tgen condef(has_xxx) contains exp(‘xxx(aaa)‘)
```

```
5   !$xev tgen list(vars) cond(has_xxx) exp
6
7   !$xev tgen var(t) exp
8       type :: t
9           integer :: dummy
10      end type t
11
12  !$xev tgen list(rem) stmt
13  !$xev tgen ctxdef(sizedefined) stmt begin
14      type(t) :: vars
15      !$xev tgen stmt(rem)
16  !$xev end tgen contextdef
17
19  !$xev tgen src context(sizedefined) begin
20      !$xev catch_size(‘xxx‘)
21  !$xev end tgen src
22  !$xev tgen dst begin
23      !$xev size_info(‘xxx‘, ‘aaa‘)
24  !$xev end tgen dst
25
26  !$xev tgen src begin
27      !$xev catch_size(‘xxx‘)
28  !$xev end tgen src
29  !$xev tgen dst begin
30      !$xev size_info(‘xxx‘, ‘1‘)
31  !$xev end tgen dst
```

The above code is a dummy infile, and `xxx` should be replaced with a real variable name via a triplet transformation.

Line 1 specifies that `xxx` is an array. Lines 3 to 5 give a standard usage of a condition: The condition in line 4 has a variable part `aaa`, which is declared in line 3. The condition must be used in a tgen-variable, which is defined in line 5. In the following rules, `vars` represents any list of expressions that contains `xxx(...)`.

Lines 7 to 10 define a user-defined type `t`, which is actually a dummy type, because the type name `t` is a tgen-variable. That is used in line 14.

Lines 12 to 16 define a context named `sizedefined`. Line 14 means that, the context is activated if there is a variable specification with any type (`type(t)`) that contains `xxx(...)` in `vars`. The context is valid in the following statements, as designated by `!$xev tgen stmt(rem)`.

Lines 19 to 24 define the transformation from `catch_size` to `size_info`. There the tgen-variable `aaa` is replaced with the array size specifier, as it is specified in line 4 as `xxx(aaa)`. For example, if there is a specification with `integer :: xxx(10, 20)`, then the transformation generates `size_info(‘xxx‘, ‘10, 20‘)`.

If there is no such a specification, then `xxx` seems to be a scalar variable, and the transformation defined in lines 26 to 31 will be activated. It inserts `1` in `size_info` directive.

5.3 Declaring a new temporary array

In the next example, we insert a declaration of a new temporary array `yyy`, which has the same dimension as obtained in `size_info`. In our real usage, we rearrange multiple dimensions into one dimension (for example, `yyy(10*20)` is generated from `xxx(10,20)`), but we omit that part for simplicity. The simplified transformation rule is as follows.

```
1   !$xev tgen list(aaa) exp
2   !$xev tgen condef(has_size) contains stmt begin
3       !$xev size_info(‘xxx‘, ‘aaa‘)
4   !$xev end tgen condef
5   !$xev tgen list(body_with_size) stmt cond(has_size)
```

Table 1: System specifications.

	Specifications				
	NEC SX-9	NEC SX-ACE	Intel Xeon E5-2630v2	NVIDIA Telsa K20	NVIDIA Tesla C2070
Type	Vector	Vector	x86	GPU	GPU
Year	2007	2013	2013	2012	2009
Peak Perf.	102.4 Gflop/s	256 Gflop/s	124.8 Gflop/s	1170 Gflop/s	515 Gflop/s
Memory Size	1000 GBytes	64 GBytes	128 GBytes	5 GBytes	6 GBytes
Memory BW	256 Gbytes/s	256 GBytes/s	51.2 GBytes/s	208 GBytes/s	150 GBytes/s
No. Cores	1	4	6	2496	448
Core Clock	3.2 GHz	1 GHz	2.6 GHz	0.71 GHz	1.15 GHz
Last-level Cache	256 KBytes	1 MBytes	15 MBytes	1.5 MBytes	768 KBytes

```
6
7   !$xev tgen src begin
8       !$xev end_of_spec
9       !$xev tgen stmt(body_with_size)
10  !$xev end tgen src
11  !$xev tgen dst begin
12      real ::  yyy(aaa)
13      !$xev end_of_spec
14      !$xev tgen stmt(body_with_size)
15  !$xev end tgen dst
```

Again, this is a dummy infile, and `xxx` and `yyy` should be replaced with names of real variables via a triplet transformation.

Lines 1 to 5 define a tgen-variable `body_with_size` that represents a list of statements containing `!$xev size_info('xxx', '...')`.

Lines 7 to 15 give a transformation to insert an array declaration. Here it is assumed that a line `!$xev end_of_spec` is inserted at the end of the specification part. It is technically possible for `xevtgen` to search for the end of the specification part, but it needs very lengthy coding. So we assume a mark that shows the end of the specification part, for simplicity and efficiency. Under that assumption, the transformation is easy. Note that the tgen-variable `aaa` conveys the size information from the `size_info` directive to the destination pattern of the transformation rule.

Here, it is assumed that `size_info` directive is placed after `end_of_spec`. Thus this is an example that information is brought from a place *after* the transformed directive, while the previous example brings information from a place *before* the transformed directive. One can write more complex transformation rules that collect information before and after the transformation point.

By using those methods, we can insert execution statements. By using 29 transformation rules, we could generate a model fitting code from a Bayesian model. That result will be reported in our future work.

6 Optimization of a real-world application with `xevtgen`

This section shows a case study of using `xevtgen` for optimizing a real-world application, called *Numerical Turbine* [8], which has been originally developed and optimized for the NEC SX-9 vector computing system installed at Tohoku University Cyberscience Center [14]. In this case study, `xevtgen` is used to migrate Numerical Turbine to other platforms so that the kernel codes are executed on the processors listed in Table 1. NEC SX compiler is used for the SX-9 and SX-ACE systems. PGI Accelerator compiler 16.4 is used for the others, and OpenACC directives [11] are used to compile the code for GPUs.

As observed in many other scientific applications, Numerical Turbine has a lot of (at least 44) similar loop nests that have almost the same loop structure shown in Figure 3(a). To achieve high performance, those loop structures must be changed so as to make better use of the system performance considering the architectural characteristics. In the original code, the length of each innermost loop is basically increased so that the SX-9 vector computing system can exploit the loop parallelism. On the other hand, it is not always best for GPUs to use the parallelism of the innermost loop. Hence, so-called loop interchange is frequently used to optimize Numerical Turbine for GPUs.

One problem is that it is difficult for the compiler to judge if each loop nest in the original code is interchangeable and parallelizable. Our preliminary evaluation indicates that those loop structures have to be altered as shown in Figure 3(b) in order to allow the OpenACC compiler to parallelize the loop nests. Note that it is necessary to insert several statements to prevent the loop interchange from changing the program behaviors. Obviously, it is difficult for compilers to automate this transformation by properly inserting the statements. Accordingly, for GPUs, all of the loop nests in the code need to be modified to make their loop structures more friendly to the OpenACC compiler.

It is effortful even for expert programmers to manually change all the 44 loop nests. In our previous work [16], one transformation rule is written in XSLT and used for transforming the loop nests in the same way as in Figure 3. The case study in [16] has demonstrated that `Xevolver` is useful to perform such repetitive code modifications by using a mechanical code transformation, which is defined by XSLT rules of only 36 code lines in the case study.

In [16], however, the users are supposed to have expert knowledge about both of performance tuning and XML technologies, even though HPC programmers are rarely familiar with both of them. On the other hand, in the case of `xevtgen`, the same rule can be represented as a dummy Fortran code of 28 code lines shown in Figure 4. This rule is general enough to transform all of the 44 loop nests whose structures are almost the same as that in Figure 3(a). For the generality, the code in lines 2 to 11 defines a list of statements, which organize a code pattern written in lines 6 to 9. The original and transformed code patterns actually defining the transformation rule are written in lines 12 to 27. This means that the rule could be simpler if the rule is used only for transforming a more specific code pattern.

According to our observation, such a simple transformation rule is often required for optimizing a specific code in practice. Such a rule could be simple but specific to a particular code, i.e., not usable for other codes, and thus requires a custom code transformation rule. In such a situation, `xevtgen` will be helpful to define and use a custom code transformation for optimizing the particular code without major code modifications.

In terms of the number of code lines, the dummy Fortran code is about 22% shorter than the XSLT rules. A more important point is that the users can write the dummy Fortran code if they know the Fortran syntax and the basic usage of `xevtgen` described in this paper. On the other hand, to describe the XSLT rules in [16], the users need to learn XML, XSLT, and ASTs generated by the ROSE compiler. Accordingly, it is obvious that the users can describe custom code transformation rules much more easily by using `xevtgen` than writing XSLT by hand.

Figure 5 shows the performance impact of the code transformation in Figure 4. The vertical axis indicates the speedup ratio of the transformed code to the original one. Thus, if the speedup ratio is less than one, the performance of the system is degraded by the code transformation.

Note that different architectures might prefer different loop structures and hence require different loop optimizations. In this case study, the original code is optimized for the SX-9 vector computing system, while the code transformation is defined for GPU-aware loop optimizations. As a result, the code transformation significantly improves the GPU performances, i.e., K20 and C2070, and degrades the performances of the others. These results clearly indicate that, if a code is simply optimized for a particular system, the optimization often leads to performance degradation of others. In the case of using code transformations, however, this performance degradation is not a problem because each system can use its own code transformation. In this particular case, the original code without any code transformation is used for the SX systems, and the transformed code is used for the GPU systems. In this way, the `Xevolver` approach can decouple "how to compute" from "what to compute" written in standard Fortran, and `xevtgen` allows users to benefit from the approach much more easily.

7 Implementation Overview

In this section, the implementation of `xevtgen` is briefly explained.

As explained in Section 2, the `xevtgen` infiles (written in Fortran) are first transformed into XML documents by `xevparse`. `Xevparse` parses a Fortran code based on `Xevolver`, also parses `!$xev` directives, and outputs an XML document. Fortran source codes should be also converted to XML documents by `xevparse`.

After reading an infile formatted in XML, `xevtgen` works in two paths. In the first path, the numbers of tgen-variable declarations, contexts, and transformation rules are counted. In the second path, `xevtgen` outputs the outfile, as an XSLT template file, while traversing the XML tree once again.

The XSLT template file has just one big transformation rule, schematically shown as follows.

```
<xsl:template match="*">
    declare XSLT parameters
    <xsl:choose>
        <xsl:when test="match source pattern">
            generate destination pattern
        </xsl:when>
        possibly more when nodes
        <xsl:otherwise>
            make a copy
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

The phrases in italics will be explained below. The last one, *make a copy* is a well-known XLST idiom that makes a copy of the current node, and recursively applies the rules to the children (see Figure 7).

We chose to use one big transformation rule with `when` branches, because we use a different method to prioritize matchings than XSLT: an earlier rule has higher priority than a later one, while in XSLT, priority is defined by numbers. It could be implemented with numbered priority, but the current implementation is more straightforward.

7.1 Declaring XSLT parameters

`Xevtgen` uses three types of XSLT parameters. First, one parameter is used for each tgen-variable. Once a node or a set of nodes is caught by a tgen-variable, it is transferred to the subsequent transformations.

Second, one string parameter is used to maintain the information about whether each node is in a context or not. Each character in this parameter corresponds to one context defined in the infile, and it is 't' if the node is within the context, and 'f' otherwise. The string is modified when a new context is found.

Third, one parameter is used for *sibling transformation*, which is not found in XSLT. Sibling transformation will be explained later in Section 7.5.

These parameters and `<xsl:choose>` are written in the outfile when the second path of `xevtgen` begins. When `xevtgen` finds a tgen-variable declaration or a condition definition, it registers those information to its internal data structure. When `xevtgen` finds a rule of `src-dst` pair or a context definition, it outputs a corresponding new `xsl:when` node. When `xevtgen` finds other kinds of XML nodes, it does nothing and keeps traversing the children and the siblings. At the end of the second phase, `xevtgen` outputs the `xsl:otherwise` node and closes the template.

7.2 Matching source pattern

One transformation rule of `xevtgen` creates a `xsl:when` node of the XSLT template. An XPath[6] expression for the `xsl:test` attribute corresponds to the source pattern of the `xevtgen` rule.

First suppose that the source pattern has no tgen-variable. Then the current node matches the source pattern when the subtree rooted by the current node is exactly the same as the subtree in the source pattern. In the current implementation, we ignore texts (which is used to indent the XML document) and comments. `Xevtgen` generates an XPath expression that requires:

- The current node has the same name as the source pattern,
- Each attribute of the source pattern has a matching attribute in the current node,
- The current node and the source pattern have the same number of attributes,
- Every child of the current node exactly matches the corresponding child of the source pattern, and
- The current node and the source pattern have the same number of child nodes.

We need to check the number of attributes (and children), not to allow a matching with a node which has more attributes (and children, respectively) than the source pattern. But if a child node is a list kind of tgen-variable, then the comparison of the numbers of children is not written to allow different numbers of children.

When `xevtgen` finds a tgen-variable in the source pattern, the corresponding XPath rule is not written. The path from the root of the source pattern to the tgen-variable is registered to an inner data structure. If the tgen-variable has conditions, an XPath rule corresponds to the conditions is written.

7.3 Generating destination pattern

When `xevtgen` reaches the destination pattern of a rule, it outputs XSLT codes to generate the destination pattern (which can be empty).

If the current node is not a tgen-variable, then a copy node with the same set of attributes is generated. Here, the children are also generated recursively.

If the current node is a tgen-variable, then the corresponding subtree is generated. If the tgen-variable is defined within the corresponding source pattern, then an XPath referring to the corresponding node is written. If the tgen-variable has `norec` specification, then a copy of the subtree is made, and otherwise, the rules are applied to the subtree with `xsl:apply-templates`.

If the tgen-variable is not defined within the corresponding source pattern, then it must be defined in a preceding transformation and be transferred via an XSLT parameter. So the reference to the corresponding XSLT parameter is generated. The transformation rules are not applied to tgen-variables transferred from preceding transformations.

7.4 Context

Context is nothing but a code transformation rule with source and destination patterns exactly the same. Not only context definition, but also a `src-dst` pair of transformation rule can be attributed as a context. The XSLT parameter corresponding to the context is modified when an `xsl:apply-templates` is invoked within the context.

7.5 Sibling transformation

In XSLT, a node or a subtree is converted to another. No mechanism is provided to transform a list of *sibling* nodes to another list of sibling nodes. For example, XSLT can convert A to $B\ C$, but does not provide a way to directly define a conversion from $A\ B$ to C, i.e., a pair of two sibling nodes A and B is converted to a different node C. We found this limitation is very inconvenient for `xevtgen`. We call a transformation rule from a list of sibling nodes into another, *sibling transformation*, and implemented it in `xevtgen`. Examples of sibling transformation are found in this paper: the context `ch1` and `ch2` in Section 4.1, and "unswitch" transformation in Section 4.2, and the rule in Section

5.3. The number of siblings is not limited to two, but for simplicity, the following descriptions assume two siblings.

The basic idea of sibling transformation is simple. To transform A B into C, a pair of transformations are defined: one to transform A followed by B to C, and another to remove B preceded by A.

There is a subtle problem in this implementation. Consider a rule to transform A B to A' B.

```
!$xev tgen var(B) stmt
!$xev tgen src begin
    A
    !$xev tgen stmt(B)
!$xev end tgen src
!$xev tgen dst begin
    A'
    !$xev tgen stmt(B)
!$xev end tgen dst
```

Here B part is caught by a tgen-variable `B`, and copied in the destination pattern. Note that `xevtgen` applies the transformation rules to the tgen-variables by default. When the rules are applied to the tgen-variable `B`, it is removed by the rule, since it is B preceded by A. Then the result is A' instead of A' B.

To solve this problem, an XSLT parameter is used to identify the applied rule. Then the rule for B preceded by A is defined to remove it only if it is not the destination pattern of the applied rule. And the rule for A is conversion into A' B, where B is not removed because it is in the destination pattern of the rule.

7.6 An example of generated XSLT template

In this section, we show output examples for the first transformation in Section 3.3 (`sqrt` example). First, the Fortran dummy code is converted to an XML document by `xevparse` as shown in Figure 6. Here, only the part of the pragma is shown, but actually it is surrounded by default Fortran preambles and postambles. This XML document should be input to `xevtgen`, and the output is as shown in Figure 7.

8 Limitations and prospective solutions

`Xevtgen` is not a complete tool, but a code transformer generator, and we are developing related tools in our toolset `Xevolver tools`. In this section, we describe some limitations of `xevtgen` and how they will be solved in the toolset we are developing.

Because `xevtgen` outputs code transformation rules in XSLT, it inherits all the limitations of XSLT. The most notable limitation in our preliminary experiences was the lack of unification. Matching between a subtree in a code and a subtree in the *infile* of `xevtgen` is possible, but one cannot compare two subtrees in the same source code. It becomes possible by creating a new *infile* referring to the source code. That generation of *infile* can be done by our toolset, since the *infile* format follows standard Fortran plus directives, which is exactly the format our toolset can do transformations. Triplet transformation reported in Section 5.1 is an example, and in our plan, it can be specified in a simpler manner with `xevdriver` and `xevutils`.

Another limitation is that, the code is transformed only from source to destination, and thus recursive application of code transformations, which plays an essential role in many code transformation systems, cannot be done solely by `xevtgen`. `Xevdriver` will control such recursive application of transformations.

In many code transformations, one has to introduce new variables. The new variable name must be different from any existing variable names. Also, sometimes one has to rename labels, as labels in a scope must be unique. Introduction of new names and new labels is, if not impossible, difficult or at least inefficient only with `xevtgen`. `Xevutils` will provide such functions. Another limitation caused by the XSLT is the lacks of transformations of names and values. We will provide a separate tool for those transformations. Also, the `xevutils` will provide information exchange among multiple files, for example, to enable interprocedural transformations.

The code transformation generated by `xevtgen` is based on the syntactic information. Extracting semantic information from codes by using `xevtgen` is technically possible, but will be complex and inefficient. In our plan, `xevutils` include some tools that embed semantic information in the code. By referring such embedded semantic information, code transformations depending on semantics will become available.

The transformations defined with `xevtgen` may not keep the semantics of the code. Preserving the semantics is not only hard to attain in general, but also unwanted in some cases: it is too restrictive. It is possible to write code transformation rules with `xevtgen` that breaks Fortran syntax, though its possibility is much more limited than direct XML transformations.

9 Related work

There are too many research works in source-to-source code transformations to list them up. By limiting the scope into Fortran, and limiting those tools that provide general source-to-source transformations (that is, not limited to a predefined set of transformations), still there are several closely related papers [12, 10, 18, 7, 1, 13]. Many of them provide deeper analysis than what `xevtgen` does, and high-level interface to predefined transformations.

An advantage of `xevtgen` compared to those related studies is its simple interface. The transformation can be defined by a dummy Fortran code with simple directives, while the other code transformation tools require a certain level of knowledge in theory and practices about compiler construction, and knowledge about some programming language (C, C++ or Java) in addition to Fortran. The simplicity of `xevtgen` might be closer to the C preprocessor, which is basically a text rewriting tool. But, unlike text rewriting tools, `xevtgen`'s transformation reflects the Fortran syntax, and it is easy to keep the transformed code following the Fortran syntax.

`Xevtgen` resembles to *syntactic macro systems*. However, there are some differences between `xevtgen` and other syntactic macro systems [9, 17, 5]. The biggest difference is that, `xevtgen` applies transforms onto particular code patterns defined in dummy Fortran codes, while in most macro systems, macros (and also C++ template) are invoked by macro identifiers. This is important for our purpose: in many cases there is already a source code of a huge number of lines, and some transformations need to be applied. Rewriting code with newly defined macros will be cumbersome and error-prone, and the maintainer of the code may not want to modify the code for a particular platform. Transformations of `xevtgen` can be applied without modifying the original code, or with little modifications such as insertions of some directives. Another difference is that, macro systems extend the language syntax, but `xevtgen` does not. So the `xevtgen` source code can be understood with the standard knowledge about the Fortran language.

The purpose of the work reported in [2] is similar to ours. They use a complex tool chain to attain code analysis and rewriting. However, their approach assumes that users already know how to use each tool of their tool chain. Thus, the users need to learn the usage of several tools. On the other hand, `xevtgen` allows users to write transformation rules in Fortran plus a small set of directives. Therefore, `xevtgen` is expected to be easier-to-learn than other tools that need special languages and/or tools for defining code transformations.

In HPC code tuning, various simple transformations are frequently needed. In some supercomputer centers, such knowledge is documented by showing examples of code rewrites, each of which consists of the code *before* rewrite and the code *after* rewrite. Sometimes it is straightforward to modify such a rewrite example into an `xevtgen` rule. Based on that observation, we expect that `xevtgen` can be used also for documentation of useful code rewrites. Actually, our collaborators[4], who have collected HPC code rewrites independently from `xevtgen`, are now trying to formulate their knowledge into `xevtgen` rules. It would be useful to collect such code rewrites in a format both human readable and machine executable.

10 Conclusion

In this paper, we have reported `xevtgen`, a code transformer generator of `Xevolver tools`. Transformations can be defined by a dummy Fortran code with some directives. Users do not need to know theory and practices of compilers to define code transformations. We have shown several examples and use cases of `xevtgen`. The implementation of `xevtgen`, which generates an XSLT template from *infile*, is briefly explained.

By using list tgen-variables and repeated applications, `xevtgen` can provide reasonably useful transformations. Some missing features will be provided by other parts of our toolset, as is discussed in Section 8. In combination with the toolset, `xevtgen` will provide the basic functionality of user-defined code transformation.

Acknowledgment

This work is partially supported by a JST CREST project "An evolutionary approach to construction of a software development environment for massively-parallel heterogeneous systems." The authors would like to thank Prof. Yamamoto of Tohoku University for allowing us to use the Numerical Turbine application. The authors are grateful also to the anonymous reviewers, whose comments enhance and improve this article.

References

[1] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program transformations for practical scalable software evolution. In *Proc. 26th International Conference on Software Engineering (ICSE'04)*, pages 625–634, 2004.

[2] Daniel Chavarria-Miranda, Ajay Panyala, Wenjing Ma, Adrian Prantl, and Sriram Krishnamoorthy. Global transformations for legacy parallel applications via structural analysis and rewriting. *Parallel Computing*, 43:1–26, 2015.

[3] Zachary Devito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erch Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based pde solver s. In *2011 International Conference for High Performance Computing , Networking, Storage and Analysis (SC'11)*, pages 9:1–9:12, 2011.

[4] Ryusuke Egawa, Kazuhiko Komatsu, and Hiroaki Kobayashi. Designing an HPC refactoring catalog toward the exa-scale computing era. *Sustained Simulation Performance 2014*, pages 91–98, 2014.

[5] Kanako Homizu, Ken Wakita, and Akira Sasaki. An implementation of a hygienic syntactic macro system for JavaScript using parsing expression grammar and a scheme macro expander. *IPSJ Transactions on Programming*, 6(2):85–101, 2013.

[6] Michael Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference (Programmer to Programmer)*. Wrox Press Ltd., 4 edition, 2008.

[7] Uwe Kuester. A language for the definition of Fortran source to source transformations. In *Computational Science and High Performance Computing IV*, pages 181–190. Springer, 2011.

[8] Satoshi Miyake, Satoru Yamamoto, Yasuhiro Sasao, Kazuhiro Momma, Toshihiro Miyawaki, and Hiroharu Ooyama. Unsteady flow effect on nonequilibrium condensation in 3-D low pressure steam turbine stages. In *ASME Turbo Expo 2013*, 2013.

[9] Hiroyasu Nagata. FORMAL: a language with a macro-oriented extension facility. *computer Languages*, 5:65–76, 1980.

[10] Boyana Norris, Albert Hartono, and William Gropp. Annotations for productivity and performance portability. In *Petascale Computing: Algorithms and Applications*, Computational Science, pages 443–462. Chapman & Hall / CRC Press, Taylor and Francis Group, 2007.

[11] OpenACC.org. *OpenACC – Directives for Accelerators*, 2011.

[12] Daniel J. Quinlan, Markus Schordan, Bobby Philip, and Markus Kowarschik. The specification of source-to-source transformations for the compile-time optimization of parallel object-oriented scientific applications. In *Proc. LCPC 2001*, pages 383–394, 2003.

[13] Georges-Andre Sibler and Alain Darte. The Nestor Library: A tool for implementing Fortran source to source transformations. In *Proc. 7th Int. Conf. High-Performance Computing and Networking (HPCN Europe 1999)*, pages 653–662, 1999.

[14] Takashi Soga, Akihiro Musa, Youichi Shimomura, Kenichi Itakura, Koki Okabe, Ryusuke Egawa, Hiroyuki Takizawa, and Hiroaki Kobayashi. Performance evaluation of NEC SX-9 using real science and engineering applications. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC09)*, pages 1–12, 2009.

[15] Reiji Suda, Luo Cheng, and Takahiro Katagiri. A mathematical method for online autotuning of power and energy consumption with corrected temperature effects. *Procedia Computer Science*, 18:1302–1311, 2013.

[16] Hiroyuki Takizawa, Shoichi Hirasawa, Yasuharu Hayashi, Ryusuke Egawa, and Hiroaki Kobayashi. Xevolver: An XML-based code translation framework for supporting HPC application migration. In *Proc. IEEE International Conference on High Performance Computing (HiPC'14)*, pages 1–11, December 2014.

[17] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proc. PLDI '93*, pages 156–165, 1993.

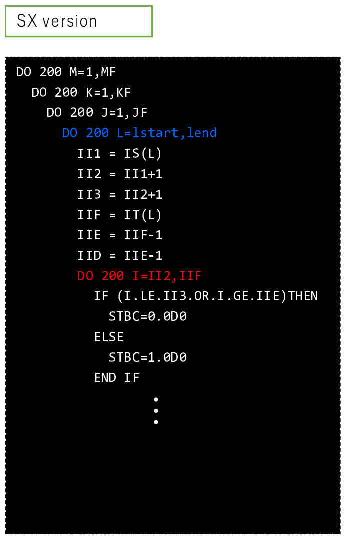
[18] Songqing Yue and Jeff Gray. SPOT: A DSL for extending Fortran programs with metaprogramming. *Advances in Software Engineering*, 2014:23, December 2014.

```
!$xev tgen list(l, l1, l2) stmt
!$xev tgen var(v) stmt
!$xev tgen var(i, i0, i1, i2) exp

!$xev tgen src begin
    !$xev loop split
    DO i = i0, i1, i2
        !$xev tgen stmt(l)
    END DO
!$xev tgen src end
!$xev tgen dst begin
    !$xev loop split begin
    DO i = i0, i1, i2
    END DO
    DO i = i0, i1, i2
        !$xev tgen stmt(l)
    END DO
    !$xev loop split end
!$xev tgen dst end

!$xev tgen src begin
    !$xev loop split begin
    DO i = i0, i1, i2
        !$xev tgen stmt(l1)
    END DO
    DO i = i0, i1, i2
        !$xev split point
        !$xev tgen stmt(l2)
    END DO
    !$xev loop split end
!$xev tgen src end
!$xev tgen dst begin
    DO i = i0, i1, i2
        !$xev tgen stmt(l1)
    END DO
    DO i = i0, i1, i2
        !$xev tgen stmt(l2)
    END DO
!$xev tgen dst end

!$xev tgen src begin
    !$xev loop split begin
    DO i = i0, i1, i2
        !$xev tgen stmt(l1)
    END DO
    DO i = i0, i1, i2
        !$xev tgen stmt(v)
        !$xev tgen stmt(l2)
    END DO
    !$xev loop split end
!$xev tgen src end
!$xev tgen dst begin
    !$xev loop split begin
    DO i = i0, i1, i2
        !$xev tgen stmt(l1)
        !$xev tgen stmt(v)
    END DO
    DO i = i0, i1, i2
        !$xev tgen stmt(l2)
    END DO
    !$xev loop split end
!$xev tgen dst end
```

Figure 1: Transformation for loop split

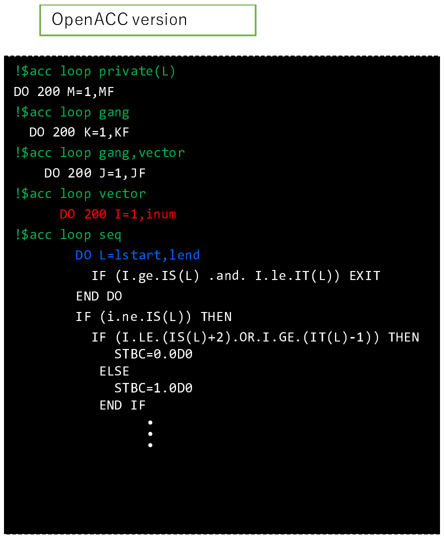
$$\begin{tikzcd}
 & \text{source code} \arrow[d] \arrow[r] & \text{destination file} \\
\text{specializing rule} \arrow[r] & {} & \text{specialized infile} \arrow[u] \\
 & \text{temporary rule} \arrow[r] & {} \\
 & & \text{dummy infile} \arrow[uu]
\end{tikzcd}$$

Figure 2: A triplet transformation



```
DO 200 M=1,MF
  DO 200 K=1,KF
    DO 200 J=1,JF
      DO 200 L=lstart,lend
        II1 = IS(L)
        II2 = II1+1
        II3 = II2+1
        IIF = IT(L)
        IIE = IIF-1
        IID = IIE-1
        DO 200 I=II2,IIF
          IF (I.LE.II3.OR.I.GE.IIE)THEN
            STBC=0.0D0
          ELSE
            STBC=1.0D0
          END IF
                 ⋮
```

(a) Loop structure in the original code.



```
!$acc loop private(L)
DO 200 M=1,MF
!$acc loop gang
  DO 200 K=1,KF
!$acc loop gang,vector
    DO 200 J=1,JF
!$acc loop vector
      DO 200 I=1,inum
!$acc loop seq
        DO L=lstart,lend
          IF (I.ge.IS(L) .and. I.le.IT(L)) EXIT
        END DO
        IF (i.ne.IS(L)) THEN
          IF (I.LE.(IS(L)+2).OR.I.GE.(IT(L)-1)) THEN
            STBC=0.0D0
           ELSE
            STBC=1.0D0
           END IF
                 ⋮
```

(b) Loop structure for the OpenACC compiler.

Figure 3: Loop nests of Numerical Turbine.

```
program nt_opt
!$xev tgen var(i1,i2,i3,i4,i5,i6,if) stmt
!$xev tgen list(body) stmt
!$xev tgen var(lstart,lend,II2,IIF) exp
!$xev tgen condef(has_doi) contains stmt begin
   DO I=II2,IIF
      !$xev tgen stmt(if)
      !$xev tgen stmt(body)
   END DO
!$xev tgen end
!$xev tgen list(stmt_with_doi) stmt cond(has_doi)
!$xev tgen src begin
   DO  L=lstart,lend
      !$xev tgen stmt(stmt_with_doi)
   END DO
!$xev end tgen src
!$xev tgen dst begin
   DO I=1,inum
      DO L = lstart, lend
         IF (I .GE. IS(L) .AND. I .LE. IT(L)) THEN
            EXIT
         END IF
      END DO
      !$xev tgen stmt(if)
      !$xev tgen stmt(body)
   END DO
!$xev end tgen dst
end program nt_opt
```

Figure 4: A dummy Fortran code for optimizing Numerical Turbine.

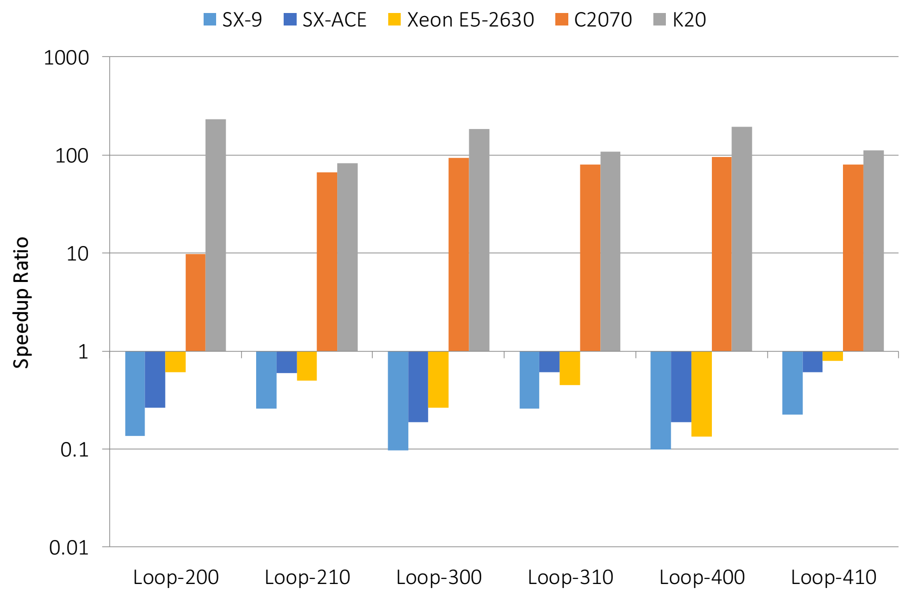


Figure 5: Numerical Turbine performance evaluation results.

```
<xev_pragma>
    <xev_clause_list>
        <xev_clause name="xev"/>
        <xev_clause name="tgen"/>
        <xev_clause name="var">
            <xev_literal name="a"/>
        </xev_clause>
        <xev_clause name="exp"/>
    </xev_clause_list>
</xev_pragma>
<xev_pragma>
    <xev_clause_list>
        <xev_clause name="xev"/>
        <xev_clause name="tgen"/>
        <xev_clause name="trans"/>
        <xev_clause name="exp"/>
        <xev_clause name="src">
            <xevparse_code_exp>
                <SgFunctionCallExp>
                    <SgFunctionRefExp name="sqrt"/>
                    <SgExprListExp>
                        <SgVarRefExp name="a"/>
                    </SgExprListExp>
                </SgFunctionCallExp>
            </xevparse_code_exp>
        </xev_clause>
        <xev_clause name="dst">
            <xevparse_code_exp>
                <SgMultiplyOp paren="0">
                    <SgVarRefExp name="a"/>
                    <SgVarRefExp name="a"/>
                </SgMultiplyOp>
            </xevparse_code_exp>
        </xev_clause>
    </xev_clause_list>
</xev_pragma>
```

Figure 6: Converted XML document from `sqrt` example by `xevparse` (Only the pragma parts are shown)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="*">
        <xsl:param name="xevtgencontext" select="'f'"/>
        <xsl:param name="xevtgensibling" select="0"/>
        <xsl:param name="xevtgenvar0" select="/.."/>
        <xsl:choose>
            <xsl:when test="count(self::*[self::SgFunctionCallExp][count(@*)=0]
                [count(*)=2][*[1]/self::SgFunctionRefExp][*[1]/@name=&quot;sqrt&quot;]
                [count(*[1]/@*)=1][count(*[1]/*)=0][*[2]/self::SgExprListExp]
                [count(*[2]/@*)=0][count(*[2]/*)=1])>0">
                <xsl:element name="SgMultiplyOp">
                    <xsl:attribute name="paren">0</xsl:attribute>
                    <xsl:apply-templates select="*[2]/*[1]/self::*">
                        <xsl:with-param name="xevtgencontext" select="'t'"/>
                        <xsl:with-param name="xevtgensibling" select="1"/>
                        <xsl:with-param name="xevtgenvar0" select="*[2]/*[1]/self::*"/>
                    </xsl:apply-templates>
                    <xsl:apply-templates select="*[2]/*[1]/self::*">
                        <xsl:with-param name="xevtgencontext" select="'t'"/>
                        <xsl:with-param name="xevtgensibling" select="1"/>
                        <xsl:with-param name="xevtgenvar0" select="*[2]/*[1]/self::*"/>
                    </xsl:apply-templates>
                </xsl:element>
            </xsl:when>
            <xsl:otherwise>
                <xsl:copy>
                    <xsl:copy-of select="@*"/>
                    <xsl:apply-templates>
                        <xsl:with-param name="xevtgencontext" select="$xevtgencontext"/>
                        <xsl:with-param name="xevtgenvar0" select="$xevtgenvar0"/>
                    </xsl:apply-templates>
                </xsl:copy>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>
</xsl:stylesheet>
```

Figure 7: Generated XSLT template from `sqrt` example by `xevtgen`