Evolving Scientific Code Adaptations with Modularization Frameworks

Pilsung Kang[1]

[1]Department of Software Science, Dankook University, Yongin, 16890, Gyeonggi, South Korea.

Contributing authors: pilsungk@dankook.ac.kr;

Abstract

Software for scientific computing has been traditionally unfavorable to changes, mainly due to its frequent use of old codebases where modern programming practices for modularity have been relatively difficult to adopt. In this paper, We present our 10-year experience of developing and evolving program adaptations in scientific software. Throughout a series of different adaptive scenario implementations, we show why managing adaptation codes using in-house tools in scientific computing has been difficult, and how we address the issue by means of modern software engineering techniques.

Keywords: software evolution, program adaptation, function call interception, aspect-oriented programming, scientific computing

1 Introduction

Software adaptation is a process of changing the behavior of a given program to achieve a certain purpose necessitated after the program's initial conception. Adapting given software usually becomes necessary due to different application requirements such as performance and stability. For instance, changing global variables and modifying functional behavior of a program module to improve application performance are typical adaptation operations. In scientific computing, adaptation is particularly relevant because scientific software often needs to change its original behavior in response to changes in the computational properties, execution environments, problem instances, and available resources, in order to ensure adequate efficiency or reliability. As described in the example scenario in Subsection 1.1, scientific computations without proper adaptation functionality can cause unsatisfactory application performance or inconsistent computational results – often a very significant waste of expensive computing resources considering that scientific applications are usually executed on a large-scale, high-performant computing environments.

However, adapting the given software to adjust its functional behavior can be hard to design, primarily due to the lack of support for modularly specifying adaptive control points and centralizing adaptive logic operations in a separate module. In addition, the need for executing adaptation without disturbing the original execution flow further complicates the adaptation process. Since directly modifying the source code to manually implement adaptations is error-prone and can often be inflexible to change for a large codebase with old software architecture [1], modular programming methods for implementing adaptive behavior without direct code modifications have traditionally been one of the main research efforts in scientific computing [2–5]. To clearly present the problem setting discussed in this paper, we illustrate a typical adaptive scenario example along with its major implementation and maintenance issues in the following section.

1.1 An Example Scenario: Dynamic Algorithm Switching

One of the most frequent adaptive scenarios in scientific computing is to replace the used algorithm with a new one at runtime, so that the computation can be performed in a more efficient or a stable manner towards convergence. Fig. 1 shows a typical algorithm switching scenario, where a linear solver method is switched to another (e.g., between stiff and non-stiff methods) following the recommender module's dynamic decision to meet certain requirements such as stability or performance. Without such adaptation functionality, the linear solver would continue to use the initially chosen method over the whole program execution, resulting in a very slow time-to-solution (with non-stiff method) or a failure to converge to a stable solution (with stiff method).

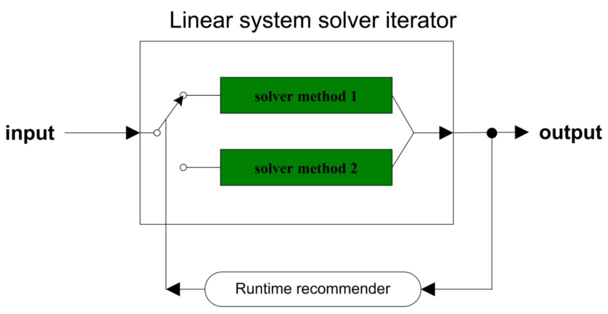

Fig. 1: An Algorithm Switching Scenario: Switching Linear Solver Methods

To implement such an adaptive scenario without directly modifying the original code, compositional approaches have mostly applied a compositional framework that integrates separately programmed adaptation implementations with the original codebase [3, 6]. The design principles of such frameworks usually include the following:

- Fine-grained control: the framework needs to provide mechanisms to access fine-grained aspects of program state or to manipulate the application behavior at the functional level, so that the original behavior can be carefully controlled to realize complex adaptive scenarios.
- Modularity: an adaptation implementation needs to be separately developed without affecting the original codebase and seamlessly combined to the original application.
- Language independence: the framework needs to support different programming languages, procedural languages in particular, and enable composition across the modules written in different languages.
- Minimal overhead: the adaptation mechanisms provided by the framework should not incur any significant overhead at runtime for the newly composed application where a separately written adaptation module is integrated to the original codebase of the application.

Previously, we also had developed a modular, compositional framework for implementing software adaptations and used it extensively to implement diverse adaptive scenarios over different scientific applications. Over time, however, the computing environments steadily kept changing and the implemented adaptations started getting broken on new settings, which required fixing and rewriting of the adaptation codes. For instance, we have seen in the last decade a rapid transition from 32-bit to 64-bit in terms of computing environments, which required rewriting for many applications across different domains. With the changes in the computing environments, however, our compositional adaptation framework became old without proper management efforts, which gradually caused it obsolete over time. We describe the adaptation management issues with our past framework in more detail in Subsection 2.3.

Due to these circumstances, we started pursuing other approaches for adapting scientific software and settled on a more solid and effective method. Based on these efforts, this paper reports the lessons we learned throughout the experiences. Specifically, we focus on scientific software adaptation in the perspectives of software management and evolution based on our previous research works [7–11], where we pursued to accomplish a modular and effective programming approach for implementing adaptive scenarios onto existing codes.

In this paper, we make the following contributions:

- We implement common adaptation scenarios in scientific computing, where very fine-grained control over the program behavior is required at the function level. To achieve the goal, we apply the compositional approach based on function call interception, thereby modularizing the adaptation code development without directly modifying the original codebase of a given application.

- We show that in-house frameworks for implementing fine-grained adaptations can often be impractical in the long term due to management issues that need to be addressed to cope with changes in the computing environments and application requirements.
- With a set of major adaptation scenarios and their implementations, we show that modern software engineering mechanisms and accompanying tools can be very useful for managing software adaptations as well as for just implementing adaptive scenarios in scientific computing where old codebases are usual.

The remainder of this paper is organized as follows. Section 2 describes our past approach based on a compositional framework and its management issues, using an example algorithm switching scenario and its implementation. Section 3 presents an advanced adaptation implementation approach based on modern software engineering techniques. Using this approach, Section 4 describes a set of different adaptive scenarios and implementations in scientific computing. Then, in Section 5, we evaluate our approach in terms of performance and programming productivity. These evaluation reports are mostly based on our past projects and experiences for implementing modular adaptations for existing scientific codes. Section 6 briefly surveys major adaptation approaches and contrasts our work with them. Finally, we summarize our work and make conclusions in Section 7.

2 Using In-house Compositional Frameworks for Implementing Adaptations

In this section, we describe our initial approach to implementing adaptations in scientific computing and its subsequent issues. Our past approach is based on a compositional framework called Adaptive Code Collage (ACC), which enabled to implement a set of diverse adaptive scenarios with scientific simulation software. We briefly present the ACC framework along with its inner workings, describe its application for implementing the algorithm switching scenario, and discuss the management issues emerged afterwards in using the in-house framework.

2.1 Adaptive Code Collage

The characteristic feature of ACC is the sophisticated use of the function call interception (FCI) technique over existing codebase for flexibly integrating separately written adaptation code to compose a new application with adaptive behavior. FCI is a technique of intercepting function calls at program runtime and have traditionally been used for application profiling and logging purposes. FCI techniques can be categorized according to the level they are

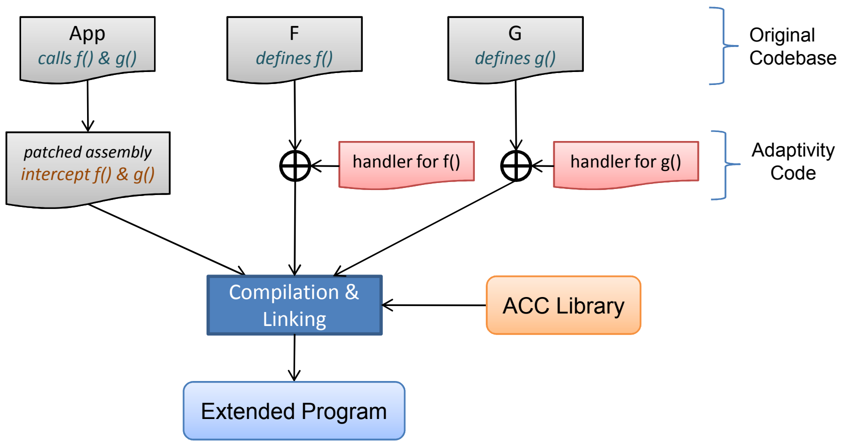

Fig. 2: Adaptation Implementation Process using the ACC framework

applied at: source language and binary object level. Source level methods need the programmer to have the original application code in hand, while binary level methods do not have such requirements. For more detailed descriptions, categorization, and applications of the FCI technique, we refer the reader to a survey such as [12].

The ACC framework internally implements FCI at the assembly language level for language-independent program composition without directly modifying the original code. Specifically, ACC intercepts at the *caller* site, where the x86 `call` instruction in the instrumentation target is replaced with a call to its own interception *handler*, a piece of code where user-defined operations can be written to realize application-specific adaptive behavior at the intercepted call.

Fig. 2 shows the adaptive scenario implementation process through ACC's FCI mechanisms. The calls to the functions such as `f()` or `g()` in the original codebase are intercepted to divert the execution control to separately written handlers. By specifying a target function to be manipulated by ACC, the programmer essentially defines an adaptive control point over the original program, where newly developed modules can be introduced to maneuver the program toward the intended adaptive behavior. Thus, as Fig. 2 shows, composition through ACC enables one to independently reason about application-specific adaptive strategies, factor them out in a centralized code, and plug in the adaptation code at control points to build an adaptive application. Besides function call interception, the ACC framework's main capabilities include function call parameter manipulation and function remapping.

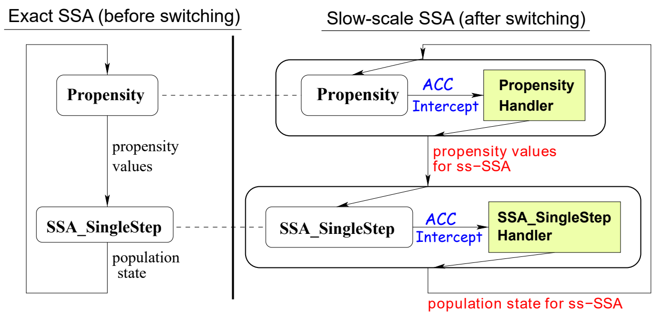

Fig. 3: Dynamic Algorithm Switching in Stochastic Simulations via ACC

2.2 ACC Approach to Implementing Dynamic Algorithm Switching

We used the ACC framework to implement a dynamic algorithm switching scenario in the biochemical simulations in our previous work [9]. Traditionally, the exact Stochastic Simulation Algorithm (SSA) [13] has been used to simulate biochemical reaction networks, which, although mathematically correct, requires significant computing power to evolve the simulated system at every time step. In the algorithm scenario, we exploited the fact that in certain reaction systems, the exact time step condition can be mitigated so that approximate but more efficient algorithms can be used without seriously harming the simulation accuracy. The *slow-scale SSA (ss-SSA)* [14] algorithm is one of such approximate algorithms, where only "slow" reaction events are simulated using the stochastic method, while the rest of the system is approximated using analytical methods. In this work, the adaptation target software package was StochKit [15], a stochastic simulation library written in C++ for simulating biochemical reaction systems on parallel environments.

Fig. 3 shows ACC's scheme for algorithm switching in a running stochastic simulation, where the relevant SSA functions in StochKit are intercepted and manipulated. In designing an adaptation scheme, it is crucial to carefully identify manipulation target functions since they work as control points where newly written adaptation code is inserted onto the original code. In this scheme, the `SSA_SingleStep` and `Propensity` functions are intercepted and modified with the associated function handlers, `SSA_SingleStep_Handler` and `Propensity_Handler` respectively, to realize the intended ss-SSA behavior.

`SSA_Singlestep_Handler` accesses the time value that is passed as the second argument of the call through ACC's parameter accessing API (application programming interface). It also accepts the user to analyze the data to determine which reaction channels are fast. `Propensity_Handler` calculates the population of the system to model the ss-SSA method while maintaining the original execution flow of the exact SSA implementation in StochKit.

Following the adaptation scheme in Fig. 3, the code in Fig. 4 implements the dynamic switching from the exact SSA algorithm to the slow-scale alternative for the fast reversible isomerization process [14]. The aforementioned two functions in the StochKit software package are intercepted and manipulated as the control points by ACC. The `SSA_Singlestep_Handler` works as the post-handler for the `SSA_Singlestep` function calls (line 4), where it calculates and substitutes the analytical solutions for the population of fast species once the user triggers the switching decision at runtime (line 12 – 14). The `Propensity_Handler` entirely replaces the `propensity` function calls by the user's decision at runtime (line 27), so that the propensity values of each reaction channel can be calculated in the slow-scale SSA fashion (line 37 – 46). As seen in this code, implementing adaptation using ACC is quite straightforward using the ACC function call interception APIs, although these take time to get used to in the fist place.

2.3 Adaptation Management Issues with the ACC Framework

.

While being effective for adapting scientific software at a detailed level, using ACC became subtly inconvenient and caused a few management issues over time. We describe the issues in this section.

2.3.1 Need for Portable Adaptations

The ACC framework's approach was based on its FCI mechanism which allowed us to manipulate function calls to tweak program behavior towards realizing adaptations. However, since ACC's FCI mechanism was implemented on 32-bit Unix environments, ACC's adaptation functionality became no longer available later on 64-bit systems, which is nowadays a standard platform for most scientific applications. This lack of portable adaptability was a major factor that led us towards more standardized software engineering solutions instead of in-house frameworks for scientific software adaptations. We need to support both 32-bit and 64-bit architectures, so that the same adaptation source code can be used to different architectures through a simple recompilation.

2.3.2 Lack of Code Reuse

In scientific programming, a certain set of adaptive scenarios occurs quite frequently. For example, changing or adjusting critical parameters of an algorithm is one of the most requested adaptive actions in scientific simulations. In fact, adaptivity schemas [16] are one of the efforts to abstract out common adaptation patterns that occur in modern scientific applications. These patterns specify the scenarios under which the execution of scientific codes can benefit from being adapted dynamically. While the ACC framework provides an effective functionality for software adaptation through the sophisticated use of

```
/* handler for original SSA_Singlestep() calls */
void SSA_SingleStep_Handler(struct acc_invoke_entry *ie, ACC_HANDLER_TYPE type)
{
  if (type == ACC_HT_POST) {     // perform as a post-handler
    if (!stopped) return;        // do nothing until user intervenes to decide switching
    if (!prop_modified) return;  // check if propensity has been modified

    Vector& xx = *origX;  // population of each species

    /* calculate population of fast species using the ranlib
       binomial distribution function */
    long xt = (long) (xx(0) + xx(1));
    xx(0) = (long) ignbin(xt, ((float) c2 /(c1+c2)));  // c: reaction rates
    xx(1) = xt -xx(0);

    prop_modified = false;
  }
}

/* handler for original propensity() calls */
void Propensity_Handler (struct acc_invoke_entry *ie, ACC_HANDLER_TYPE type)
{
  if (type == ACC_HT_PRE) { // perform as a pre-handler
    if (!stopped) return;   // do nothing if not stopped for switching

    /* replace with our version of propensity calculation */
    acc_remap(ie, (acc_invoke_func_t *)MyPropensity);
  }
}

/* propensity function for slow-scal SSA */
Vector MyPropensity(const Vector& x)
{
   Vector& xx = *origX;    // population of each species

   /* use approximate, analytical solution for fast species */
   long xt = (long) (xx(0) + xx(1));
   double xx1_inf = xt * (c1/(c1+c2));   // c: reaction rates

   Vector copyX = xx;
   copyX(1) = xx1_inf;

   /* calculate propensity for slow species */
   Vector a = Propensity(copyX);
   for (int i=0; i<NCHANNELS; i++) {
     a(i) = a(i) * (1 - propensity_zeros[i]);
   }

   /* set the flag to relax the fast variables right after exact SSA
    * has been performed */
   prop_modified = true;

   return a;
}
```

Fig. 4: Dynamic Algorithm Switching Implementation using ACC: Switching from Exact SSA to Slow-scale SSA in Biochemical Reaction Network Simulations

FCI, it lacks the ability to codify patterns of recurring adaptations at the high level of expressibility in scientific simulations. As a result, using ACC was not very favorable to reuse adaptation codes and the programmer needs to write application-specific adaptation code for every adaptive scenario without code reuse.

3 Aspect-Oriented Programming for Scientific Software Adaptations

In this section, we describe a modern software engineering technique, known as Aspect-Oriented Programming (AOP), which we adopted instead of the ACC framework in order to implement scientific software adaptations. We also

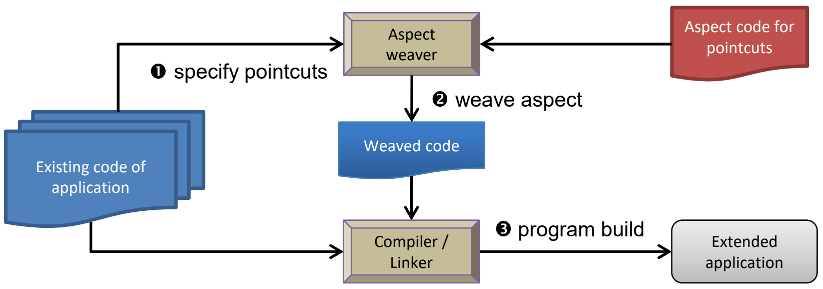

Fig. 5: Adaptation Implementation Process using AOP frameworks

describe the benefits of using the AOP constructs for adapting and maintaining scientific software by presenting practical applications of using AOP for implementing adaptive scenarios.

3.1 Separation of Concerns: Securing Modularity and Maintainability

Aspect-Oriented Programming (AOP) [17] is a programming paradigm in software engineering which attempts to help programmers in the software design process through the principle of *separation of concerns*. There are aspects of software programs, such as logging, security, and performance, that cannot be easily captured by typical inheritance hierarchy of object-oriented programming techniques. AOP helps to modularize those concerns spread across the whole program into *cross-cutting concerns*. Applying this paradigm, we consider software adaptation as a separate concern in our approach, allowing us to modularize adaptation code externally with regard to the main codebase. The newly written *aspect* code (i.e., adaptation code) is then combined to the original code through the AOP framework to compose an adaptive scientific application. Adopting this modern software engineering mechanism secures improved modularity and maintainability.

To realize adaptive scenarios around the scientific software codebase, we use a C++ AOP extension, called AspectC++ [18]. During the advice weaving process, AspectC++ performs source-to-source translation on an aspect code, and generate standard C++ code. This allows us to work on both 32-bit and 64-bit systems without any portability issues. The translated C++ code is fed to a normal C++ compiler along with a target application code to form an FCI-enabled executable at last.

3.2 Adaptation Implementation Process via AOP

A major way of implementing a cross-cutting concern in AOP is to use *advice*, which is a piece of code to be performed at intended program execution points specified as a *pointcut*. When a running program reaches a *join point*, the location in an application where specified pointcuts match, and if the point is at a function or method call, the call is intercepted and the program control is transferred to an associated advice code for execution. This process of instrumenting an advice code into a target program at associated pointcuts is called *advice weaving*, which does not involve directly modifying the original codebase. AOP languages offer programming constructs to express pointcuts and advice code, which can be invoked before or after executing a target method. In addition, an advice can entirely replace the original target call.

Fig. 5 illustrates a general process of application development in the AOP environment. An adaptive scenario is considered as a separate concern and its implementation code is factored out in an aspect code. The written aspect code is then combined with the original code across pointcuts through the aspect weaver to compose an application with adaptive behavior.

3.3 Patterns of Adapting Scientific Applications

As a branch of the object-oriented programming (OOP) paradigm, AOP enables to model real-world objects as classes and encourages code reuse by means of inheritance between classes (i.e., aspects). As to AspectC++, it enables to express adaptability functionality as aspects. Furthermore, well-known adaptation patterns can be expressed as abstract aspects and can be reused via aspect inheritance across different adaptive applications in scientific computing.

Adaptation patterns frequently found in scientific computing includes algorithmic parameter change and algorithm switching. The algorithmic parameter change pattern represents the cases where the execution behavior of a scientific computation can be affected by adjusting the algorithm's parameters to better match the dynamic characteristics of the computational progress. The dynamic algorithm switching pattern, as described in Section 1.1, represents the cases where an algorithm used in a running simulation is switched to another one to meet certain execution requirements such as simulation stability or performance. Later in Section 4.2, we present an aspect implementation of the algorithmic parameter change pattern in AspectC++ and demonstrates the effectiveness of code reuse by using the AOP based adaptation approach.

3.4 Fortran Support

Fortran has been one of the mainstream languages in the scientific computing community and there are a large portion of Fortran application codebase. Hence, supporting Fortran applications in adaptive programming is essential. To achieve this goal, our ACC framework supported Fortran, and other languages as well, by instrumenting the code at the assembly level. In contrast, our new AspectC++ approach employs C wrappers for Fortran functions to interface with C++ aspects, as AspectC++ uses source-to-source translation for weaving. In this scheme, we expose as C wrappers those portions of Fortran code that need to directly interact with adaptation code written in AspectC++. We developed a C wrapper generator by extending the F2PY [19], a Fortran to Python interface generator, and used it to automatically generate C functions having the compatible signatures with the wrapped Fortran functions. Then, the AspectC++ weaver can combine adaptation advice code with the original Fortran programs via the **`execution`** pointcuts.

3.5 Limitations

In the implementation perspective, our AspectC++ based approach currently supports adaptations written in compiled imperative languages such as C, C++, and Fortran. Interpreted languages such as Java or Python are not supported, in large part due to the fact that compiled languages have traditionally been in the mainstream for programming scientific software for maximal performance without any overhead incurred by using the interpreter layer. However, our approach can be applied in a straightforward manner to interpreted languages, especially Java where sophisticated modularization mechanisms including AOP are quite rich. For instance, AspectJ [20, 21] supports weaving of a piece of new Java code onto existing codebase both at compilation time and program runtime, thus effectively composing a newly adapted version of the given application.

Another drawback of our method is that the source code of the target software needs to be available to apply adaptation, which may not be always the case depending on different situations. Our approach is based on source-to-source transformation of a given application program, which is then further processed by compilation onto the native execution platform.

4 Adapting Scientific Software via AOP: Real-world Examples

In this section, we demonstrate the effectiveness of using the AOP-based approach by showcasing a set of real-world adaptive scenario examples and their implementations using the AOP techniques. The examples include dynamic algorithm switching, algorithmic parameter change, and substitution of library modules.

4.1 Dynamic Algorithm Switching using AspectC++

This example scenario and its implementation using ACC were described in Section 2, where the exact SSA algorithm switches to the slow-scale algorithm in simulating biochemical reactions at the user's runtime decision. We show an equivalent implementation using AspectC++ here, as reported in our previous work [10]. Fig. 6 shows the AspectC++ aspect code for algorithm switching in our application. At $t = STOPTIME$, the simulation stops to accept the user's input, switching to the ss-SSA early in the simulation (line 5). At line 11, to intercept the `SSA_SingleStep` calls, we use the `before` advice at the **`execution`** pointcut of `SSA_SingleStep`. At line 13 and 14, we fetch the current system state (i.e., population of each species) and simulation time

```
 1 /* AlgoSwitchSSA.ah - algorithm switching aspect for SSA.cpp in StochKit */
 2 #include <mpi.h>
 3 #include "Vector.h"
 4
 5 extern double STOPTIME;
 6 bool algo_switched = false;
 7 int propensity_zeros[NCHANNELS];
 8
 9 aspect AlgoSwitchSSA {
10   /* intercept the call to SSA_SingleStep using before advice */
11   advice execution("% ...::SSA_SingleStep(...)") : before() {
12     /* fetch arguments for system vector and current simulation time */
13     CSE::Math::Vector *origX = (CSE::Math::Vector*) tjp->arg(0);
14     double *pTime = (double *) tjp->arg(1);
15
16     if (!algo_switched) { // algorithm not switched
17       if (*pTime > STOPTIME) {
18         int rank, size;
19         rank = MPI::COMM_WORLD.Get_rank();  // get current process id
20         size = MPI::COMM_WORLD.Get_size();  // get number of processes
21
22         /* if root process, prompt the user with current system states */
23         if (rank == 0) {
24           showSimulationStates();
25           getFastReactionsFromUser();
26           }
27         }
28         /* broadcast info about fast reactions and algorithm switching */
29         MPI::COMM_WORLD.Bcast(propensity_zeros, NCHANNELS, MPI::INT, 0);
30         algo_switched = true;
31         MPI::COMM_WORLD.Bcast(&algo_switched, 1, MPI::BOOL, 0);
32       }
33     }
34     else {  // algorithm switched
35       CSE::Math::Vector& xx = *origX;
36       updatePopulation(xx);  // calculate and update system states
37     }
38   }
39 };
```

Fig. 6: Aspect Code for Switching from SSA to Slow-scale SSA in AspectC++

passed as the first and the second argument of `SSA_SingleStep` using the AspectC++ `tjp->arg()` function parameter access API. That is, like the ACC framework, manipulating function call parameters can be easily performed in AspectC++.

If the used algorithm has not been switched (line 16) and the simulation pause time has reached (line 17), the simulation stops for the MPI (message-passing interface) [22] root process to show the current state of the simulation (line 24) and to prompt the user for information about possible fast reactions (line 25). The information provided by the user is stored in `propensity_zeros` and broadcast across the MPI environment to all the participating processes (line 29 and 31), which will switch to the ss-SSA algorithm for the remaining simulation using the broadcast information. Once the algorithm is switched to the ss-SSA, the aspect code computes an analytical solution for the population of each fast species identified by the user, and updates the system states accordingly (line 36). For the propensity function, we implement a simple AspectC++ code as previously done in the ACC approach, where the calls to the original propensity function are intercepted and changed to realize the ss-SSA style simulation.

Fig. 7 shows the execution time measurement results for the dynamic algorithm switching implementation of the stochastic simulation, where the fast reversible isomerization process with a single slow reaction was considered [14]. The experiments were performed on a Linux cluster of Intel Xeon E5-2470v2

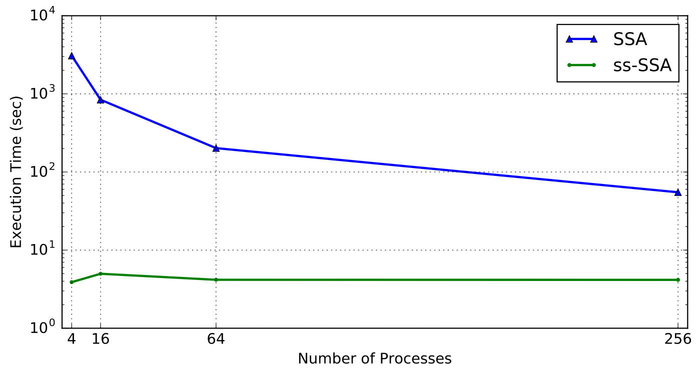

Fig. 7: Execution time of algorithm-switched ss-SSA simulation with varying number of nodes, in comparison to execution time of exact SSA (4096 samples for fast reversible isomerization process) [10]

systems, each with 96GB memory and 80GB SSD, interconnected with 10Gbit Ethernet. As seen in the figure, the performance gain from algorithm switching is significant compared to just sticking with the exact SSA algorithm, As an example, with 4 processes producing 1,024 samples each, the algorithm switching scenario consumed only 3.9 seconds per process, whereas the exact SSA took almost an hour to complete all 1,024 samples on each process, thus resulting in almost three orders of magnitude in performance improvement.

4.2 Adjusting Algorithmic Parameters using AspectC++

This example is about changing algorithmic parameters at runtime, which is one of the most typical adaptive scenarios in scientific computing. Specifically, we implement an adaptive successive over-relaxation (SOR) algorithm for linear solvers. For many iterative solvers, the convergence rate is heavily influenced by parameters, and in this SOR case, it is the over-relaxation parameter, ω. Although there are well known techniques in numerical libraries to automatically adapt the ω parameter, most of them are based on a simple heuristic that works "reasonably" well for a range of matrices, without taking advantage of the problem-specific information that may be available in a given problem. By using modular adaptation techniques, we can implement a problem-specific parameter adjustment scenario, and in this case, we implement an adaptive SOR with an external parameter change code written in AspectC++.

Since the algorithmic parameter change scenarios is very popular, and also because code reuse via inheritance is well supported in AOP, we can patternize it as an abstract code for general cases in AspectC++. And later this abstract

```
 1  aspect ControlSystems {
 2  public:
 3    // pointcut at system state function
 4    pointcut virtual checkSystemState() = 0;
 5
 6    // adjust system parameters
 7    virtual void adjustParam(void* state) = 0;
 8
 9    // adjust parameters to adapt to system state changes
10    advice execution(checkSystemState()) : after() {
11      void* state = tjp->result();
12      adjustParam(state);
13    }
14  };
```

Fig. 8: Control Systems Schema: Abstract Aspect for Algorithmic Parameter Change in AspectC++

```
 1  aspect SORParamControl: public ControlSystems {
 2    // SOR iterator as the control point
 3    pointcut checkSystemState() = "% __wrap_itsor_(...)";
 4
 5    // override adjustParam to implement adaptive SOR
 6    void adjustParam(void* state) {
 7        double new_omega, omega;
 8
 9        // ...
10        // code for calculating the new omega value
11
12        setParam(new_omega);
13    }
14
15    // set a new SOR parameter value
16    void setParam (double param) {
17        set_omega_(&new_omega);
18    }
19  };
```

Fig. 9: Adjusting the ITPACT SOR Parameter via Inheriting Abstract Aspect in AspectC++

aspect code can be inherited to implement specializations for individual applications. As described in Section 3.3, this is one of the benefits of using modern software engineering tools such as AOP instead of in-house tools.

Such an abstract base code is shown in Fig. 8 as the `ControlSystems` aspect. The adaptive control points are specified as the `checkSystemState()` pointcut (line 4), so that the execution control is transferred to the adaptation advice at this pointcut (line 10). Inside the advice, intermediate computation results are examined (line 11) and the algorithmic parameter is adjusted appropriately by `adjustParam()` (line 12). The `adjustParam()` function is declared as **`virtual`** (line 7), so that it can be inherited later as a specialization for a given application.

Fig. 9 shows an AspectC++ implementation for an adaptive SOR solver with an automatic ω adjustment, which inherits and extends the `ControlSystems` abstract aspect in Fig. 8. Specifically, the adaptation target is the SOR routine in the ITPACK package [23], a 40–year old Fortran numerical library. Since ITPACK is written in Fortran, we expose the pointcut function (`itsor_()` as a C wrapper (`__wrap_itsor_()` at line 3) to interface with the AspectC++ aspect code, as described in Section 3.4. Then, the **`after`**`()` advice as defined in the abstract aspect will be automatically performed at the specified pointcut, where the overriding `adjustParam()` is executed to examine the current computation state and to adjust the over-relaxation parameter ω in response to the changes (line 12). By expressing the common adaptive scenario as an abstract aspect and reusing it through inheritance, we can save the code while reducing the complexity in the development of an application-specific adaptation code in a modular fashion. Specifically, we were able to obtain smaller code size for the adaptation logic implementation of the adaptive SOR, which amounts to 20% of code reduction in the AspectC++ version over the previous ACC framework version.

4.3 Substitution of a Library using AspectC++

In addition to dynamic algorithm switching and algorithmic parameter change, there is often a need for library substitution in scientific computing. For example, stochastic simulations are highly dependent upon the choice of the random number generator (RNG) since RNGs directly affect the simulation speed and quality. Therefore, programmers often want to use a more recent or optimized version of the RNGs. However, adopting a new software library is often a non-trivial job in most modern, complex software systems because switching to a new library involves rewriting of the original program, which may cause unwanted human errors. In such cases, adaptive programming with modular composition tools such as the AOP frameworks can be a simple and affordable way of adopting a new software library onto existing codebase, where no direct modification of the original source code is needed.

Previously, we had implemented library substitution using the ACC framework [24]. And later, we employed the AspectC++ mechanism and updated the work [11]. The following is a brief description of the AspectC++ based implementation.

```
/* Switch to different RNGs using AspectC++ advice */
aspect Substitute_RNG {
  /* Replace C random() calls via around() */
  advice execution("long random(void)") : around() {
    long retval;

#ifdef USE_SPRNG        /* replace with SPRNG */
    retval = (long) isprng();

#elif defined(USE_SFMT) /* replace with SFMT */
    retval =  (long) (gen_rand32() >> 1);

    /* Substitute return value */
    long *result = (long *) tjp->result();
    *result = retval;
  }
};
```

Fig. 10: Substitution of Random Number Generation Libraries in StochKit using AspectC++

Fig. 10 shows our AspectC++ code to switch to another RNG library in the StochKit software package, which implements a diverse set of stochastic simulation algorithms for simulating biochemical reaction systems. In our implementation, we intercept the calls to the C `random()` function by using AspectC++ `around()` advice (line 4), which entirely replaces the original C `random()` calls. Depending on the macro definition at compile time, our implementation chooses between the SPRNG [25] library (line 7) and the SIMD-based Mersenne Twister (SFMT) [26] library (line 10). The random number generated by the selected RNG library is returned to the original calls to the C `random()` function through the AspectC++ return value manipulation API (line 14 and 15). Overall, the library substitution scenario has been effectively implemented by means of the function call interception capability of the AspectC++ framework and its language constructs for expressing the concern separately in an aspect code.

5 Evaluation

5.1 Performance Overhead

We have observed the performance overhead of our AspectC++ adaptation approach to be quite small, as reported in our previous works [7, 11]. The overhead of our approach is mainly caused by function call catching at runtime and it ranges from 0.1% to 0.8% depending on applications and execution environments. The effect of overhead can become reduced further particularly for such adaptation scenarios where a large amount of computations are performed at a single function call, thus making the overall impact on application performance by function call interception virtually negligible.

5.2 Productivity via Reusable Aspects

Code reusability is an important software design objective that allows the same code to be used in different scenarios, improving programmer productivity and reducing code size and maintenance burden. In OOP, inheritance is a technique that promotes code reusability by encapsulating common functionality in a base class and extending it with subclasses. AOP also allows for improved programming productivity via aspect code inheritance, which is one of the principal benefits from adopting modern OOP techniques.

In the following, we briefly present one of our past evaluation results in adapting scientific software using our AOP-based adaptation approach, focusing on code reusability through inheriting from aspect base code for implementing application-specific adaptation scenarios [7]. For instance, we previously described about 20% of code reduction in Subsection 4.2.

The target software we used for productivity evaluation is GenIDLEST [27], a computational fluid dynamics (CFD) simulation application written in Fortran to solve the Navier-Stokes and energy equations. Adaptation scenarios include timestep parameter adjustment, fluid flow model switching (dynamic algorithm switching), and parameter search space mining. Table 1 compares the number of uncommented lines of source code (ULOC) between AspectC++

	ACC			**AspectC++**		
Adaptation	aux	logic	total	aux	logic	total (gain)
Timestep control	17	31	48	9	20	29 (40%)
Flow model switching	20	68	88	13	53	66 (25%)
Parameter mining	25	172	197	13	154	167 (15%)

Table 1: ULOC comparison of the GenIDLEST adaptation implementations between the ACC and AspectC++ implementations [7]

and ACC-based implementations in each of these scenarios. The "aux" category includes code that is not directly related to the adaptation "logic" such as header includes and helper functions. The ACC-based implementations also require the use of the ACC framework's APIs to introduce adaptation code to GenIDLEST.

AspectC++ versions require less code than ACC-based implementations, achieving 15% to 40% reduction in code size. Using AspectC++ results in fewer lines for adaptive functionality. ACC-based implementation requires extra code for ACC framework instantiation. Inherited abstract aspects in AspectC++ lead to fewer hand-written lines. Join point specifications impact code reduction effectiveness, favoring subclassing abstract aspect base code for complex adaptation schemes with multiple join points.

6 Related Work

Function call interception (FCI) is a low-level mechanism frequently used for changing an application's behavior. There is a variety of FCI constructs and techniques implemented at different levels of program development, ranging from the instruction level [28–30], to the source code compilation level [31–33], and to the linking and loading level [34–36]. For a detailed survey of FCI and its applications in computing, we refer the reader to [12].

In high-performance computing on cluster environments, there is a large body of research studies for supporting program adaptation at the level of middleware or runtime platforms [5, 37, 38]. The main goal of these efforts is to realize adaptation at the middleware for efficient utilization of the execution environment through load-balancing or application task scheduling. Therefore, the adaptation policies are usually coarse-grained based on resource availability and external execution parameters, which implies that function level adaptations are hardly considered. This is in contrast with the adaptation granularity and its implementation level that our work pursues, where fine-grained aspects of program behavior can be adapted at the function level based on the intermediate computation results as well as the changes in the execution environment.

Domain-specific languages (DSLs) coupled with automated code generation techniques such as metaprogramming [39] are one of the recent approaches for managing and evolving legacy codes in scientific computing [40]. For instance, Yue and Gray [41] develop a DSL called SPOT to realize automated transformations of existing Fortran programs. One of the major design goals of their DSL is modularization of the adaptation code development with respect to the original codebase. To achieve this goal, similarly to our AOP based approach, they consider adaptations as cross-cutting concerns across the codebase and separately implement adaptive scenarios. The designed DSL is used to specify a variety of different adaptations including code profiling, parallelization, and checkpointing for Fortran applications.

There are a couple of advantages in using a DSL. One of the major strengths of adaptation approaches based on metaprogramming is the ability to generate or translate code for program transformation. Once the programmer specifies an intended adaptation using the syntax of the given DSL, a program transformation engine (PTE) translates the DSL code into an ordinary code that conforms to the standard of the target programming language such as C++ or Fortran. This is quite similar to our AOP-based approach, where adaptation is described in an aspect language and the aspect code is first translated to an ordinary code, which is then compiled to the native code by the standard compiler. Another advantage of using a DSL is greater expressibility towards a specific, targeted software development purpose, which is natural considering the motivation of using a DSL instead of general purpose languages.

There are some drawbacks in using a DSL. Most of all, introducing a DSL to a software development project requires a significant amount of buy-ins including the efforts to learn a whole new language consisting of entirely new syntax, constructs, and semantics, which can be burdensome to programmers. This is in contrast to using AOP languages, which are usually a small extension to popular programming languages. In addition, to use a DSL usually entails adoption of a separate PTE for translating a DSL code. For instance, the SPOT language [41] uses a PTE called ROSE [42] as a backend translator for SPOT. In this respect, our AOP-based approach can be a more lightweight and accessible option for implementing scientific software adaptation, compared to DSL-based approaches.

7 Conclusions

In this paper, we presented a modular approach to expressing adaptation scenarios based on modern software engineering solutions. Our long-term experiences with real-world applications show that the AOP-based approach is more effective than in-house frameworks in terms of portability across different architectures, which improves maintainability over a long period of time. Like other compositional approaches, the AOP-based approach allows the adaptive functionality to be implemented externally to the main codebase, so that the original application code and the adaptation code can be maintained independently. We also showed that even in scientific programming, using AOP language constructs allows for more opportunities of code reuse via aspect inheritance, which is one of the foremost benefits of adopting the object-oriented program paradigm.

Overall, the capability of separating out adaptation concerns independently and the expressibility of adaptation aspects provided by the AspectC++ framework with its language constructs can be very useful for implementing common adaptive scenarios in scientific computing. Hence, facing the growing challenges of modern scientific applications, we expect that the need for sophisticated software engineering mechanisms will become more and more significant in managing and evolving scientific software adaptations.

Acknowledgments

The present research was supported by the research fund of Dankook University in 2022.

8 Declarations

8.1 Ethical Approval

Not applicable.

8.2 Competing Interests

The authors declare no competing interests.

8.3 Authors' Contributions

Not applicable.

8.4 Funding

The present research was supported by the research fund of Dankook University in 2022.

8.5 Availability of Data and Materials

The tool used in this work is AspectC++ and it is publicly available in its website. A replication code bundle is available at https://github.com/pilsungk/AdaptiveSOR. Other target software packages are too old and currently unavailable.

References

[1] Lehman, M.M., Belady, L.A. (eds.): Program Evolution: Processes of Software Change. Academic Press Professional, Inc., San Diego, CA, USA (1985)

[2] Voss, M.J., Eigenmann, R.: High-level Adaptive Program Optimization with ADAPT. In: PPoPP '01: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, pp. 93–102. ACM, New York, NY, USA (2001)

[3] Yu, H., Zhang, D., Rauchwerger, L.: An Adaptive Algorithm Selection Framework. In: PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pp. 278–289. IEEE Computer Society, Washington, DC, USA (2004)

[4] Du, W., Agrawal, G.: Language and Compiler Support for Adaptive Applications. In: SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, p. 29. IEEE Computer Society, Washington, DC, USA (2004)

[5] Janjic, V., Hammond, K., Yang, Y.: Using Application Information to Drive Adaptive Grid Middleware Scheduling Decisions. In: MAI '08: Proceedings of the 2nd Workshop on Middleware-Application Interaction, pp. 7–12. ACM, New York, NY, USA (2008)

[6] An, P., Jula, A., Rus, S., Saunders, S., Smith, T., Tanase, G., Thomas, N., Amato, N.M., Rauchwerger, L.: STAPL: An Adaptive, Generic Parallel C++ Library. In: Dietz, H.G. (ed.) LCPC, pp. 193–208. Springer, London, UK (2001)

[7] Kang, P., Tilevich, E., Ramakrishnan, N., Varadarajan, S.: Maintainable and Reusable Scientific Software Adaptation: Democratizing Scientific Software Adaptation. In: Proceedings of the Tenth International Conference on Aspect-Oriented Software Development (AOSD '11), pp. 165–176 (2011). Proceedings of the tenth International Conference on Aspect-Oriented Software Development (AOSD '11)

[8] Kang, P., Heffner, M.A., Ramakrishnan, N., Ribbens, C.J., Varadarajan, S.: Adaptive Code Collage: A Framework to Transparently Modify Scientific Codes. IEEE Computing in Science and Engineering **14**(1), 52–63 (2012)

[9] Kang, P.: Modular Implementation of Dynamic Algorithm Switching in Parallel Simulations. Cluster Computing **15**(3), 321–332 (2012)

[10] Kang, P.: Dynamic Algorithm Switching in Parallel Simulations using AOP. Journal of Information Science and Engineering **34**(6), 1367–1382 (2018)

[11] Kang, P.: Implementing Adaptive Decisions in Stochastic Simulations via AOP. IEICE Trans. Information and Systems **E101-D**(7), 1950–1953 (2018)

[12] Kang, P.: Function Call Interception Techniques. Software: Practice and Experience **48**(3), 385–401 (2018)

[13] Gillespie, D.T.: A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions. J. Computational Physics **22**(4), 403–434 (1976)

[14] Cao, Y., Gillespie, D.T., Petzold, L.R.: The Slow-Scale Stochastic Simulation Algorithm. Journal of Chemical Physics **122**(1), 014116 (2005)

[15] Li, H., Cao, Y., Petzold, L.R., Gillespie, D.T.: Algorithms and Software for Stochastic Simulation of Biochemical Reacting Systems. Biotechnology Progress **24**, 56–61 (2008)

[16] Varadarajan, S., Ramakrishnan, N.: Novel Runtime Systems Support for Adaptive Compositional Modeling in PSEs. Future Gener. Comput. Syst. **21**(6), 878–895 (2005)

[17] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming, vol. 1241, pp. 220–242. Springer, Berlin, Heidelberg, and New York (1997)

[18] Spinczyk, O., Gal, A., Schröder-Preikschat, W.: AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In: CRPIT '02: Proceedings of the 40th International Conference on Tools Pacific, pp. 53–60. Australian Computer Society, Inc., Darlinghurst, Australia (2002)

[19] F2PY: Fortran to Python interface generator. http://cens.ioc.ee/projects/f2py2e/

[20] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming, pp. 327–353. Springer, London, UK (2001)

[21] AspectJ http://www.eclipse.org/aspectj [accessed on March 1, 2023]

[22] Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA (1999)

[23] Kincaid, D.R., Respess, J.R., Young, D.M., Grimes, R.R.: ITPACK 2C: A FORTRAN Package for Solving Large Sparse Linear Systems by Adaptive Accelerated Iterative Methods. ACM Trans. Math. Softw. **8**(3), 302–322 (1982)

[24] Kang, P., Cao, Y., Ramakrishnan, N., Ribbens, C.J., Varadarajan, S.: Modular Implementation of Adaptive Decisions in Stochastic Simulations. In: Proceedings of the 24th Annual ACM Symposium on Applied Computing, pp. 995–1001. ACM, ??? (2009). Proceedings of the 24th Annual ACM Symposium on Applied Computing

[25] Mascagni, M., Srinivasan, A.: Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation. ACM Trans. Math. Softw. **26**(3), 436–461 (2000)

[26] Saito, M., Matsumoto, M.: SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator. In: Monte Carlo and Quasi-Monte Carlo Methods, pp. 607–622 (2006)

[27] Tafti, D.: GenIDLEST - A Scalable Parallel Computational Tool for Simulating Complex Turbulent Flows. In: Proceedings of the ASME Fluids Engineering Division (FED), vol. 256. ASME-IMECE, ??? (2001)

[28] Hunt, G., Brubacher, D.: Detours: Binary Interception of Win32 Functions. In: Proceedings of the 3rd USENIX Windows NT Symposium, Seattle, Washington, USA, pp. 135–144 (1999)

[29] Bruening, D.: Efficient, Transparent, and Comprehensive Runtime Code Manipulation. PhD thesis, MIT (September 2004)

[30] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 190–200. ACM Press, New York, NY, USA (2005)

[31] Program Instrumentation Options, GNU Compiler Collection. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

[32] Schaefer, C.A., Pankratius, V., Tichy, W.F.: Atune-IL: An Instrumentation Language for Auto-tuning Parallel Applications. In: Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing, pp. 9–20. Springer, Berlin, Heidelberg (2009)

[33] Charif-Rubial, A.S., Barthou, D., Valensi, C., Shende, S., Malony, A., Jalby, W.: MIL: A Language to Build Program Analysis Tools Through Static Binary Instrumentation. In: 20th Annual International Conference on High Performance Computing, pp. 206–215 (2013)

[34] Kiselyov, O.: Patch–free User-level Link-time Intercepting of System Calls and Interposing on Library Functions. http://okmij.org/ftp/

syscall-interpose.html (2008)

[35] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. Concurrency and Computation: Practice and Experience **22**(6), 685–701 (2010)

[36] Luu, H., Behzad, B., Aydt, R., Winslett, M.: A Multi-level Approach for Understanding I/O Activity in HPC Applications. In: 2013 IEEE International Conference on Cluster Computing (CLUSTER), Indianapolis, IN, USA, pp. 1–5 (2013). IEEE

[37] Jérémy Buisson and Francoise André and Jean-Louis Pazat: A Framework for Dynamic Adaptation of Parallel Components. In: Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005, vol. 33, pp. 65–72 (2006)

[38] Baude, Françoise and Henrio, Ludovic and Ruz, Cristian: Programming Distributed and Adaptable Autonomous Components – the GCM/ProActive Framework. Software: Practice and Experience **45**(9), 1189–1227 (2015)

[39] Spinellis, D.: Rational Metaprogramming. IEEE Software **25**(1), 78–79 (2008). https://doi.org/10.1109/MS.2008.15

[40] Murai, H., Sato, M., Nakao, M., Lee, J.: Metaprogramming Framework for Existing HPC Languages Based on the Omni Compiler Infrastructure. In: 2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW), pp. 250–256 (2018). https://doi.org/10.1109/CANDARW.2018.00054

[41] Yue, S., Gray, J.: OpenFortran: Extending Fortran with Metaprogramming. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'13) (2013)

[42] Rose Compiler – Program Analysis and Transformation. http://rosecompiler.org [accessed on March 1, 2023]