Implementing Adaptive Decisions in Stochastic Simulations via AOP

Pilsung KANG\(^{1(a)}\), Member

1. Introduction

The programmer of scientific software is often faced with various simulation options such as algorithmic alternatives and different libraries. Although a decision can be made in such cases to meet simulation requirements or to improve time-to-solution, the afterwards course of action usually involves modifying existing source code to include any newly decided simulation option. However, source code modifications to implement such adaptive decisions can be a serious burden at times since necessary code changes can be spread across the whole program structure and manual rewriting of software often induces human errors.

In this paper, we present a modular approach to implementing adaptive decisions in scientific simulations using modern software engineering mechanisms. Specifically, we employ the machinery of aspect-oriented programming (AOP)\(^{[1]}\), a programming paradigm that enables the programmer to express concerns crossing the entire program within one location (called aspects in AOP). By adopting AOP, our approach secures modularity in developing adaptive functionality onto an existing program.

This paper improves our previous work\(^{[2]}\) where we employed an in-house software framework for implementing adaptive decisions. A major drawback of using in-house tools is that such tools are not generally available to the programming community in the domain, which greatly harms the applicability of the programming method. Through the use of AOP tools and mechanisms which are more accessible and popular, we increase the applicability and generality of our approach.

The remainder of this paper is organized as follows. Section 2 provides the background information for our work. Section 3 describes adaptive scenarios in performing simulations using real-world scientific software, and the implementation details of our approach using AOP. Section 4 evaluates our AOP-based approach in terms of performance and code quality. Finally, we summarize our work and make conclusions in Sect. 5.

2. Background

In this section, we briefly describe the context of our work. We provide a short overview of stochastic simulation of biochemical networks because we use a stochastic simulation program as the target example of our approach. We also describe concepts and programming constructs of AOP.

2.1 Stochastic Simulation of Biochemical Networks

In the biochemistry domain such as cell biology, simulating a system of different species and their reactions can take two different approaches. The ODE (ordinary differential equations) solving approach models the evolution of the system as a set of ODEs and numerically solves the system of equations. In contrast, the stochastic approach models the system with the dynamical state vector of the species and the reaction specifications among the species (called reaction channels). Gillespie’s Stochastic Simulation Algorithm (SSA)\(^{[3]}\) is central in the stochastic approach. In the SSA, the system is evolved by randomly choosing one reaction to fire and by generating random samples for the chosen reaction channel for the next time interval.

In our work, we use StochKit\(^{[4]}\) for stochastic simulation and we implement a set of adaptive decisions onto the existing code base of StochKit, whose core simulation components consist of 198 C/C++ source files, totaling about 37K uncommented lines of code.

2.2 AOP Concepts and Constructs

In the AOP paradigm, a newly emerged concern for an existing application is separately written and managed in the form of advice, a piece of code to execute at intended control points in program execution. The control points are called join points in the AOP vernacular, and are specified...
by pointcut expressions. When program execution reaches a join point, the execution control is transferred to the advice code, such that the advice operations are performed to take appropriate actions that are planned by the programmer. For instance, when a function is specified as a pointcut, the calls to the function is dynamically intercepted to execute associated advice code for the function. The advice code can execute before or after the function call, and even can entirely replace the call. Integrating advice code to a given program is called advice weaving in AOP, and it is processed by the weaver or weaving compiler provided by the AOP toolchain. AOP weaving can be traditionally performed statically [5], [6] or dynamically [7], [8]. In addition, modern languages with managed runtime environments like Java and C# support load-time weaving [9] as a special type of dynamic weaving.

In our approach for implementing adaptive scenarios with scientific applications, we use AspectC++ [10] because traditional imperative languages are still more popular in the scientific programming domain.

3. Adaptive Scenarios and AOP Implementations

We present two adaptive scenarios in the context of using StochKit and their corresponding implementations using AspectC++. These examples show adaptive decisions can be implemented in a modular fashion where direct code modification is avoided. Specifically, these scenarios are implemented by effectively replacing a subprogram module in an application with an alternative module without directly modifying original code. Thus, our implementations can be considered as an adequate interim solution to updating a software package without an explicit version update.

3.1 Change of Algorithmic Parameter Values

Algorithmic parameters are critical in running a scientific simulation because they directly affect the simulation performance and stability. Considering that days or weeks of running time are not unusual in scientific simulations, rightly-chosen parameter values can significantly improve time-to-solution in scientific computing. However, algorithmic parameter values are sometimes hardwired onto different parts of code and manually modifying them is often cumbersome and also can lead to human errors. Rather than directly touching the source code, the effect of changing the values can be obtained by replacing the subprogram module enclosing hardwired parameters with another module that uses different parameter values. To replace a subprogram module without direct source code modifications, intercepting existing function calls is essential and the entire process of function interception and substitution can be implemented through AOP advice weaving.

One example of hardwired parameter values in StochKit is the PoissonRandom function in Random.cpp as shown in Fig. 1. This function generates random numbers from the Poisson distribution for the τ-leaping method [11], for which StochKit provides 7 differently optimized versions. As shown in line 5 and 7 in the code, the StochKit implementation of hardwired parameter value change is performed statically. In line 5, the AspectC++ execution pointcut matches a PoissonRandom function returning double in any C++ namespace. When matched, we use the around construct to execute the advice code in place of the PoissonRandom function (line 5). To access the mean value passed as an argument of PoissonRandom, we use a parameter fetching API of AspectC++, tjp->arg(0). Then, using the fetched mean value, we approximate a random number from the Poisson distribution using adjusted parameter values, $10^{-6}$ and $10^{-4}$, in the branches for different approximations of the Poisson distribution depending on the size of mean.

Using AOP’s advice weaving, the hardwired values in the original function can be changed seamlessly by entirely replacing the function. Figure 2 shows our aspect implementation of changing the parameter values in PoissonRandom. In line 4, the AspectC++ execution pointcut matches a PoissonRandom function returning double in any C++ namespace. When matched, we use the around construct to execute the advice code in place of the PoissonRandom function (line 5). To access the mean value passed as an argument of PoissonRandom, we use a parameter fetching API of AspectC++, tjp->arg(0). Then, using the fetched mean value, we approximate a random number from the Poisson distribution using adjusted parameter values, $10^{-1}$ and $10^{-3}$. The approximated value is set as the return value of the original PoissonRandom calls through the AspectC++ return value manipulation API, tjp->result(). The new parameter values (i.e., $10^{-1}$ and $10^{-3}$) in the aspect code are chosen as an extreme case to distinctly demonstrate the effect of the change in the simulation. Using AspectC++’s ag++ weaving compiler, our aspect code is transparently combined with the existing StochKit program, thus resulting in an application that executes our advice code at the calls to StochKit’s PoissonRandom function.
To evaluate our AspectC++ implementation, we consider the stiff decaying dimerization process [12]. In fact, StochKit bundles an example code of this process in the distribution package. The biochemical system in the dimerization process consists of three species $S_1$, $S_2$, and $S_3$, and the reactions are described as the following equations.

\[
S_1 \xrightarrow{c_1} 0, \quad S_1 + S_1 \xrightarrow{c_2} S_2, \quad S_2 \xrightarrow{c_3} S_3.
\]

For the reaction rate constants, we use the default values in StochKit: $c_1 = 1, c_2 = 0.002, c_3 = 0.5$, and $c_4 = 0.04$. The initial populations of the species are $S_1 = 10,000$, $S_2 = 0$, and $S_3 = 0$, respectively, and the final simulation time is set to 10. Applying our AspectC++ implementation for changing the parameter values, we simulate 10,000 samples of the system.

Table 1 shows the simulation results of our AspectC++ implementation of PoissonRandom along with the original version of StochKit. While the mean values of the species are almost identical between the two versions, the standard deviations are significantly small for the changed version of PoissonRandom. This is an intended behavior for our demonstration purposes because in our version, the Poisson random value is very roughly approximated in most cases due to the tiny values of the parameter, instead of being more exactly calculated. Every call to the original PoissonRandom function is intercepted to transfer the execution control to our aspect advice code that uses new values of the parameter in calculating Poisson random numbers.

3.2 Substitution of Random Number Generator Libraries

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 up-to-date or optimized version of the RNGs by switching to a newer version of the RNG library. However, adopting a new software library is often a non-trivial job in most modern, complex software systems because applying a new library can involve rewriting of the original program such as including new header files. In such cases, using AOP’s modularized programming techniques can be a simple and affordable way of including a new software library onto existing code base, where no direct modification of the original source code is needed.

Figure 3 shows our AspectC++ implementation of switching to different RNGs at the calls to the C random function in StochKit. The aspect code chooses between the SPRNG [13] library bundled in StochKit and the Mersenne Twister SFMT [14] library, if available, depending on the C macro definition at compile time. The scheme used in the aspect implementation is virtually the same as the previous example. We use the AspectC++ execution pointcut to catch the random calls and execute our around advice code to replace the calls (line 4). We make the calls to SPRNG (line 8) or SFMT (line 11) depending on the macro definition. Then, we substitute the random number from the chosen RNG library for the original random calls (line 14 and 15).

4. Evaluations and Considerations

Performance Overhead

To evaluate the performance overhead caused by the switching functionality in our AspectC++ adaptation implementations, we perform dimerization process simulations for 10,000 samples on a 64bit Linux system (kernel 3.10) with 2x Intel Xeon E5-2470v2 10 core @ 2.40Ghz (i.e., 20 cores and 40 hardware threads in total), 96GB Main memory @ 1600MT/s, and 80GB SSD system drive. For comparison, we measure the execution time for 3 different implementations: manually modified version, adaptation implementation by the in-house tool in our previous work [2], and the implementation by AspectC++. We note that the StochKit distribution and all the adaptation implementations were built as 32 bit executables because the in-house tool supports only 32 bit environments. In addition, the execution time measurements were performed 5 times for each approach and the results were averaged over the 5 simulation runs.

Table 2 shows the average execution time to run the StochKit simulation for each of the manual-rewrite, the in-house tool, and the AspectC++-based adaptation approaches for the hardwired algorithmic parameter change application. In this adaptation example, the overhead is caused by intercepting the PoissonRandom call at the callee site and accessing the passed argument at the call. As can be seen in the table, the performance cost of AspectC++-based adaptation is quite small, incurring 0.8% on the system. This overhead is comparable to that incurred by the implementation using the in-house tool from our previous work (0.75% overhead). Considering that there are about 10,000 PoissonRandom calls for simulating a single sample with the final time set to 10, approximately 3ns were spent...
on each call.

Code Quality
Software development using AOP typically organizes cross-cutting concerns within separate code so that modularity of program components is enabled. Improved code savings and reduced code complexity are usual through the use of sophisticated pointcut expressions and inheritance. However, our approach in using AOP is an extension of our previous work [2], where we seek an effective method of implementing simulation adaptation scenarios in a modular fashion, so that original application behavior is changed without explicitly altering the original code structure. In this regard, like our previous approach, code savings from using AOP in our applications is not significant compared with manually modified versions. In addition, code complexity of our AOP implementations is essentially the same as our previous approach, because the logical structure of the adaptation implementations is unchanged. However, unlike our previous approach where an in-house tool is used, our AOP approach here is more effective and generally applicable because AOP constructs are greatly acceptable and the AspectC++ tools are widely accessible.

5. Summary and Conclusions

In this paper, we presented a modular approach to implementing adaptive decisions in scientific simulations. Our approach applies the AOP paradigm to consider adaptive decisions as a separate concern in software development, thus enabling better software management where direct modification of the original source code is avoided. Through the applications of our approach to real-world stochastic simulation software we demonstrated the applicability of our approach. AOP is well known for its modularity benefits in software development and in the future, we expect that AOP or similar programming mechanisms will be increasingly useful in scientific programming.

Acknowledgements

This work was supported by a 2018 research grant from Youngsan University, Republic of Korea.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin, “Aspect-Oriented Programming,” ECOOP’97 — Object-Oriented Programming: 11th European Conference, vol.1241, pp.220–242, 1997.
[2] P. Kang, Y. Cao, N. Ramakrishnan, C.J. Ribbens, and S. Varadarajan, “Modular Implementation of Adaptive Decisions in Stochastic Simulations,” Proceedings of the 24th Annual ACM Symposium on Applied Computing, pp.995–1001, March 2009.
[3] D.T. Gillespie, “Exact Stochastic Simulation of Coupled Chemical Reactions,” The Journal of Physical Chemistry, vol.81, no.25, pp.2340–2361, 1977.
[4] H. Li, Y. Cao, L.R. Petzold, and D.T. Gillespie, “Algorithms and Software for Stochastic Simulation of Biochemical Reacting Systems,” Biotechnology Progress, vol.24, no.1, pp.56–61, 2008.
[5] E. Hilsdale and J. Hugunin, “Advice Weaving in AspectJ,” Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, pp.26–35, 2004.
[6] P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. De Moor, D. Sereni, G. Sittampalam, and J. Tibble, “abc: An Extensible AspectJ Compiler,” Transactions on Aspect-Oriented Software Development I, Lecture Notes in Computer Science, vol.3880, pp.293–334, Springer, 2006.
[7] A. Popovici, G. Alonso, and T. Gross, “Just-in-time Aspects: Efficient Dynamic Weaving for Java,” Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, AOSD ’03, New York, NY , USA, pp.100–109, 2003.
[8] A. Vasseur, “Dynamic AOP and Runtime Weaving for Java - How Does AspectWerkz Address It?,” Proceedings of the Dynamic Aspect Workshop, Lancaster, England, pp.135–145, March 2004.
[9] AspectJ Load-Time Weaving, https://www.eclipse.org/aspectj/doc/released/devguide/ltw.html.
[10] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, “AspectC++: An Aspect-Oriented Extension to the C++ Programming Language,” CRPIT ’02: Proceedings of the 40th International Conference on Tools Pacific, Darlinghurst, Australia, pp.53–60, 2002.
[11] D.T. Gillespie, “Approximate Accelerated Stochastic Simulation of Chemically Reacting Systems,” The Journal of Chemical Physics, vol.115, no.4, pp.1716–1733, 2001.
[12] M. Rathinam, L.R. Petzold, Y. Cao, and D.T. Gillespie, “Stiffness in Stochastic Chemically Reacting Systems: The Implicit Tau-Leaping Method,” The Journal of Chemical Physics, vol.119, no.24, pp.12784–12794, 2003.
[13] M. Mascagni and A. Srinivasan, “Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation,” ACM Trans. Math. Softw., vol.26, no.3, pp.436–461, 2000.
[14] M. Saito and M. Matsumoto, “SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator,” Monte Carlo and Quasi-Monte Carlo Methods, pp.607–622, 2006.

Table 2 Execution time (seconds) and overhead measurements of the adaptation implementations for simulating the dimerization process.

| Scenario       | Manual Rewrite | Modular Adaptation |
|----------------|----------------|--------------------|
|                | In-house Tool  | AspectC++ Weaving  |
| Parameter Change | 38.01          | 38.29 (.75%)       |
|                | 38.31 (.80%)   |                    |