AN ASPECT-BASED APPROACH FOR CONCURRENT PROGRAMMING USING CSP FEATURES

José Elias Araújo, Henrique Rebêlo, Ricardo Lima, Alexandre Mota, Fernando Castor, Tiago Lima
Juliana Lucena and Filipe Lima
Informatics Center, Federal University of Pernambuco, Recife, Brazil

Keywords: Concurrent programming, Aspect-oriented programming, Communicating sequential processes, JCSP, Metrics.

Abstract: The construction of large scale parallel and concurrent applications is one of the greatest challenges faced by software engineers nowadays. Programming models for concurrency implemented by mainstream programming languages, such as Java, C, and C++, are too low-level and difficult to use by the average programmer. At the same time, the use of libraries implementing high level concurrency abstractions such as JCSP requires additional learning effort and produces programs where application logic is tangled with library-specific code. In this paper we propose separating concurrent concerns (CSP code) from the development of sequential Java processes. We explore aspect-oriented programming to implement this separation of concerns. A compiler generates an AspectJ code, which instruments the sequential Java program with JCSP concurrent constructors. We have conducted an experiment to evaluate the benefits of the proposed framework. We employ metrics for attributes such as separation of concerns, coupling, and size to compare our approach against the JCSP framework and thread based approaches.

1 INTRODUCTION

Java usually relies on thread based mechanisms to control concurrency. This is too low level and requires much effort from programmers with solid background on concurrent programming to develop simple and small concurrent applications. Moreover, working on thread-level makes the program difficult to debug and error prone. Based on these observations, Welch proposed the Java CSP (JCSP) (Welch, 2006) framework, which adopts CSP constructs to create concurrent Java programs. JCSP's programmers do not rely on threads or concurrency patterns like *future* and *oneway* (Goetz et al., 2006). Welch claims that JCSP loses some ultra-low process management overheads but wines the model for a mainstream programming language. Unfortunately, concurrency features and sequential processes are still intertwined (tangled) in JCSP programs.

One might see the JCSP concurrency constructs that span multiple sequential modules as a crosscutting concern. Exploring the methodology introduced by aspect-oriented programming (AOP), we could introduce a new unit of modularization by implementing the JCSP concurrency features as an aspect from AOP. An *aspect weaver*, which is a compiler-like entity, would automatically instrument the sequential code with concurrency aspects to compose the final system. In this paper, we explore these ideas to propose a new concurrency programming style for Java programs, named Aspect-Oriented JCSP - AJCSP. We essentially annotate sequential Java programs with concurrency specifications using a CSP-like syntax. We created a compiler to translate such annotations into an AspectJ code, which is responsible for instrument a sequential Java program with JCSP code.

This paper also presents quantitative assessments of two systems implemented in three different versions: AJCSP, JCSP, and Java threads. Our study was based on well-known software engineering attributes such as separation of concerns, coupling, and size. We have found that our aspect-oriented solution with AJCSP improved the separation of concurrency concern. In addition, we have observed that the use of AJCSP: (i) decreased the coupling between the concurrent and the sequential code; (ii) can help to decrease code scattering, improving the modularity; (iii) is useful to remove tangling of concurrency concern, enhancing program readability, and (iv) reduced the number of attributes, operations, and lines of code in

Elias Araújo J., Rebêlo H., Lima R., Mota A., Castor F., Lima T., Lucena J. and Lima F. (2010).
AN ASPECT-BASED APPROACH FOR CONCURRENT PROGRAMMING USING CSP FEATURES.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 226-231
DOI: 10.5220/0003014102260231

```
Producer = |~| data: {1..100} @ ch!data -> Producer
Consumer = ch?x -> print!x -> Consumer
Program = Producer || Consumer
```

Figure 1: Producer and consumer in CSP.

a particular system due to the "aspectization" of concurrency.

The remainder of this paper is organized as follows. Section 2 presents the existing framework to implement concurrency control. Section 3 presents our approach to implement JCSP features using AOP. Section 4 presents a quantitatively assessment of the impact of our AJCSP approach in a case study involving two systems. Also, study results in terms of separation of concerns, coupling, and size attributes are presented. Finally, Section 5 includes some concluding remarks and directions for future work.

2 JAVA CSP

JCSP (Welch, 2006) is a Java library developed by Professors Peter Welch and Paul Austin from the Univesity of Kent at Canterbury. It is based on the formal specification language CSP (Roscoe et al., 1997; Hoare, 1985) created by Tony Hoare in 1975 and updated by Bill Roscoe in 1998, serving as an implementation medium for this language. CSP basic elements are events, processes and operators on events and processes. Processes are behavioral units that exchange data using the message passing model of communication instead of the more traditional shared memory model supported directly in Java as discussed in Section 2.1.

Since JCSP is an object-oriented language, it encapsulates the implementation details necessary to support the elements of CSP, in terms of an API which provides concepts as channels, processes and operators. Thus, the translation from CSP to JCSP is almost direct except for the treatment of recursion and some CSP capabilities not supported by the library; for instance, multiway-rendezvous (multiple synchronization).

A key factor of using the combination between CSP and JCSP is that one can model a system using CSP, analyze its main properties, such as deadlock, livelock and determinism via the CSP model checker FDR (Hoare, 1985), and after being satisfied with the model, translate it in terms of JCSP. This allows an almost guaranteed implementation in terms of the previous checked properties.

In what follows, we provide a brief overview about JCSP using a simple example written in CSP as well as in JCSP. With this, we expect the reader can see that the translation is almost direct.

The example shown in Figure 1 is a classical producer-consumer program written in CSP. This example illustrates a synchronized parallelism between two processes. The Producer process is responsible for generating a random values from 1 to 100 and then outputting this value through the channel ch. The Consumer process waits for the random value using an input communication via channel ch. After receiving this value, the process sends it to its environment via channel print.

Based on Figure 1, we can propose a JCSP version capturing an equivalent behavior. As JCSP needs more details to work properly we need three figures: Figure 2(a) shows the JCSP version of the Consumer process and the Producer process, and Figure 2(b) of the parallel combination between the Consumer and Producer processes.

We start our JCSP presentation with the consumer-producer system depicted in Figure 2. This figure illustrates the Producer process (lines `1` to `10`). The first thing to note about a JCSP process is that it is a traditional Java class except for the need to implement the interface org.jcsp.lang.CSProcess. As a class, we have attributes, a constructor and one or more methods, where the method `run()` is mandatory. Our single attribute is a channel, named `ch` (line `2`), used to output values (this is characterized by the keyword ChannelOutput). The constructor simply initializes the output channel with the single parameter with the same type. The method produce() captures the CSP construction |~| **data: 1..100 @** as a function which returns a random value from 1 to 100. Finally, the method `run()` represents the CSP recursion by an infinite loop, whose behavior is basically sending the result of the method `produce()` using the output channel `ch`.

Similarly to the Producer, the Consumer (lines `12` to `21`) class implements the interface org.jcsp.lang.CSProcess and provides a run() method to be implemented. The ch attribute is used to input values. Thus, this process begins reading a value available in the ch channel and store it in another attribute named data. The method `print()` captures the CSP construction `print!x` which sends the value bound to x to its environment via channel print. It also represents the CSP recursion by an infinite loop, where its behavior is to print values received from channel ch.

The final element of our JCSP version is the main behavior. It is also given by a Java class but now it does not implement interface CSProcess. Class Program captures the parallelism between processes

```
public class Producer  implements CSProcess  {
    private ChannelOutput ch;

    public Object produce(){ ...}

    public void run() {
        while (true) {
            Object data = produce();
            ch.write(data);
        }
    }
}
public class Consumer  implements CSProcess  {
    private ChannelInput ch;

    public void print(Object object){ ...}

    public void run() {
        while (true) {
            Object data = ch.read();
            print(data);
        }
    }
}
```

(a)

```
public class Program  implements CSProcess {

    public void run() {
        One2OneChannel ch = Channel.one2one();
        Producer producer =new Producer(ch.out()) ;
        Consumer consumer =new Consumer(ch.in()) ;
        Parallel parallel =new Parallel();
        parallel.addProcess(producer);
        parallel.addProcess(consumer);
        parallel.run();
    }
    public static void main(String[] args){
        new Program(). run() ;
    }
}
```

(b)

Legend: ▨ Concurrency Concern Code

Figure 2: (a) Producer and Consumer classes implemented with JCSP. (b) The main Program class with JCSP.

Producer and Consumer. As the class Program simply links the previous processes, we use a channel typed One2OneChannel. After that, we create instances of processes Producer and Consumer and use these instances to create a group of independent (parallel) processes. To execute the elements inside this group, the method `run()` is called (Welch, 2006). The result of Program is an infinite sequence of random based communications between Producer and Consumer.

3 ASPECT-ORIENTED JCSP - AJCSP

In this section we describe AJCSP programming style as well as some important points connected to the compiler implementation.

3.1 The AJCSP Programming Style

In AJCSP, the concurrency annotations are inserted inside Java comments. It uses a Java single line comment (`//`) followed by the symbol @# to indicate that the line contains AJCSP annotations. We call such annotations `class prefix`. Since they are Java comments, AJCSP annotations are ignored if the conven-

```
//@# var data
//@# data = produce() -> ch!data -> Producer
public class Producer {
   public Object produce(){...}
}
//@# var data
//@# ch?data -> print(data) -> Consumer
public class Consumer {
   public void print(Object data){...}
}
//@# Producer[||] Consumer
public class Program {
   public static void main(String[] args
        ){
      (new Program()). run() ;
   }
}
```

Legend: ▨ Concurrency Concern Code

Figure 3: The Producer/Consumer implementation using AJCSP approach.

tional Java compiler is used. In this case, the compiler generates a sequential Java program.

AJCSP processes are Java classes. Such processes are composed through AJCSP constructs to define a concurrent Java program.

3.2 An Example: Producer-consumer

In Section 2.2, we illustrated the implementation of a producer-consumer concurrent problem using JCSP. This section describes an AJSCP version of the same problem.

Classes `Producer` and `Consumer` in Figure 3 are pure Java classes. No JCSP code is used to implement these classes. The AJCSP annotations are employed to specify the concurrent behavior processes instantiated from such classes. The reader may compare them with the classes in Figure 2. Notice that the latter classes implements the JSCP interface CSProcess. Moreover, CSP constructs are mixed with the Java code used to implement the behavior of processes `Producer` and `Consumer.`

In Figure 3 (line 2), the method `produce` is invoked and return a value of type Object, which is stored in variable `data.` Afterwards, the process `Producer` writes the `data` value in the channel `ch.` At this moment, `Producer` is blocked until another process `(Consumer)` reads the value sent through channel `ch.` Then, `Producer` is recursively invoked. On the other hand, process `Consumer` reads the channel `ch` and waits until another process `(Producer)` writes in the channel `ch. Consumer` then prints that value and is recursively called.

The `Program` class (see Line 12 in Figure 3) represents the starting point of the Producer-Consumer application, I define the parallel composition of the Producer and Consumer objects (processes) (see Line 11 in Figure 3).

4 CASE STUDY

In this section, we present a case study conducted to evaluate the benefits and limitations of AJCSP when compared against JCSP and Java threads.

4.1 Study Settings

In this subsection, we describe the configuration of our case study. In particular, we discuss the goals and the research questions we intend to investigate. Finally, we discuss the metrics suite employed in our study as well as our assessment procedures.

Table 1: Separation of Concerns metrics results.

System	Version	CDC	CDO	LOCC	DOSC	DOSM	DOTC
ProdCons	Threads	4	8	62	0.92	0.94	0.83
	JCSP	3	6	44	0.99	0.93	0.73
	AJCSP	3	0	6	1	0	0.67
Bingo	Threads	4	10	21	0.95	0.88	0.38
	JCSP	3	7	50	0.97	0.85	0.80
	AJCSP	3	0	6	0.97	0	0.15

Goal. The main goal of the case study is to assess whether AJCSP contributes to produce concurrent code of higher quality, when compared against JCSP and Java threads. Notice that such assessment is based on modularity. Hence, we are concerned about issues like: (i) scattering of concurrency concern; (ii) tangling between the concurrency concern and the sequential program.

Research Questions. We investigate seven research questions in the case study. Which approach contributes to decrease: (RQ1) scattering of the concurrency concern?; (RQ2) tangling between the concurrency concern and the sequential (business) code?; (RQ3) the number of components?; (RQ4) coupling between components?; (RQ5) the lines of code in components?; (RQ6) the lines of code in components?, and (RQ7) the number of attributes and operations in components?

Metrics. In order to answer the research questions, we selected a metrics suite proposed in (Sant'anna et al., 2003) to evaluate separation of concerns, coupling, and code size These metrics were adapted form classic OO metrics (Chidamber and Kemerer, 1994) to be applied to the AOP paradigm. We discuss the employed metrics in the rest of this section.

Lower values for a given metrics implies better results, for instance the two versions of scattering metrics we use (DOSC and DOSM) varies from 0 (completely localized) to 1 (completely delocalized, present in all components).

Separation of Concerns (SoC) metrics measure the degree to which a single concern (concurrency control in our study) affects the system. The coupling metric CBC indicates the degree of dependency between components. Excessive coupling is not desirable, since it is detrimental to modular design. Size metrics are important to evaluate the complexity of the final system. For further details about size metrics, refer to (Chidamber and Kemerer, 1994).

Assessment Procedures. We implemented three versions (using AJCSP, JCSP, and Java threads) of two different applications: Producer and Consumer (ProdCons); and the bingo game (Bingo). We implemented

Table 2: Size and coupling metrics results.

System	Version	LOC	NOA	NOO	VS	CBC
ProdCons	Threads	74	4	11	4	5
	JCSP	52	2	9	3	7
	AJCSP	22	0	5	3	0
Bingo	Threads	190	10	26	5	7
	JCSP	174	14	23	4	11
	AJCSP	138	6	20	4	3

the same functionalities for each version of the applications. This is important to perform a fair comparison.

In the measurement process, the data was partially gathered by the AJATO measurement tool [1]. It supports some metrics: LOC, NOA, NOO. Additionally, we used the AOP metrics tool [2] to collect CBC and VS. Eventually, we collected the SoC metrics (CDC, CDO, LOCC, DOSC, DOSM, and DOTC) (Sant'anna et al., 2003; Eaddy et al., 2008) manually.

4.2 Study Results

This subsection presents the results of the measurement process. The data have been collected based on the set of defined metrics. The presentation is organized in two parts. First, we describe the results for the separation of concerns metrics. Then, we present the results for the size and coupling metrics.

4.2.1 Separation of Concerns Measures

Table 1 shows the results for the SoC metrics. The AJCSP versions of the target systems performed better than the other two versions. The application of the SoC metrics was useful to quantify how effective was the separation of the concurrency control concern in the target systems. In relation to the measure of CDC, all target systems present similar results for the three implemented versions. By considering the VS metric from Table 2, we can observe (using the CDC metric) that the concurrency concern crosscuts almost the components of the target systems in all versions. Thus, we can conclude that none of three versions of the target systems provides a good separation of concern regarding scattering. Code scattering is one symptom that indicates that a system fails to modularize a particular concern that is implemented in multiple components (classes) (Laddad, 2003). Since all versions demonstrated to be too scattered in relation to their components, we employed the DOSC metric. With the DOSC, we can quantify exactly the degree of how scattered the concurrency concern is in each version of the target systems. After measurement, we realized that the AJCSP version presented the worst results in implementing the concurrency control by its components. This is due to the annotative approach imposed by AJCSP.

Even though the AJCSP approach presents the worst results for code scattering across components, our approach is superior for both CDO and DOSM metrics. We employed such metrics to quantify and measure the degree of how scattered the concurrency concern is in relation to all operations in the target systems. We observed that except our approach, the other two (threads and JCSP) presented higher CDO and DOSM. In fact, the AJCSP implementation presents 0 for both CDO and DOSM. This divergence is a direct consequence of the strategy we adopted for annotating the concurrency behavior in classes. Since we put all annotations before a class definition, we decouple the methods (operations) from the concurrency concern code. As a result, we have a more legible code which implements only the business concern. By considering the four metrics for scattering measurement, the AJCSP approach is the answer for the first research question (RQ1).

Code tangling is another symptom of non-modularization of a particular concern (Laddad, 2003). Code tangling is caused when a component handles multiple concerns simultaneously. Hence, to measure the degree of tangling of the concerns (concurrency and business) implemented by the three versions of the target systems, we employed the DOTC metric. The AJCSP approach showed to be less tangled when compared with the other approaches. We can easily notice that by reasoning in how the AJCSP approach is achieved. So, since we concentrate the concurrency concern code as annotations in the beginning of a class, this way, the rest of the code is dedicated only to the business concern implementation. As a result, the code became less tangled with the concurrency control code. The main reason is that the concurrency concern code appears in several methods of a class in the other approaches. This can observed in the CDO and DOSM metrics previously discussed. As a consequence of our approach, the tangling degree is *inversely proportional* to a system size (LOC metric in Table 2). Thus, the greater the lines of code of a system is, the lower degree of tangling it is. Therefore, this answer the second research question (RQ2).

Finally, regarding LOCC (Lines of Concern Code) metric, we can observe that since AJCSP concentrates the concurrency concern implementation into a single place, it requires less lines of concern code to implement such a concern in contrast to the other ap-

[1]http://www.teccomm.les.inf.puc-rio.br/emagno/ajato/
[2]http://aopmetrics.tigris.org/

proaches. This properly answer the third research question (RQ3).

4.2.2 Size and Coupling Measures

We have also analyzed how the AJCSP implementation version has impacted positively or negatively on the size and coupling measures in comparison with its counterparts in Thread and JCSP. Table 2 presents the results for these metrics for both AJCSP and evolved system versions. The use of the annotative approach of AJCSP led to a reduction of all size metrics (Table 2). For example, in the Bingo system, the LOC metric in AJCSP were, respectively, 27% and 20% lower than its counterparts in Thread and JCSP. Moreover, AJCSP version of Bingo system showed less NOA (40% and 57%, respectively) than Thread and JCSP versions. Note that we had similar results for the ProdCons system. (This answer the research questions RQ5 to RQ7).

The AJCSP solution in Bingo system was superior to its counterpart solutions in terms of coupling. The coupling (CBC) in the AJCSP implementation was 57% and 73% lower than Thread and JCSP, respectively. The coupling was too lower in AJCSP when compared to JCSP because the code of latter is completely dependent of the JCSP API. Since we abstract the use of such API, we provided a significant reduction in the coupling metric. This is one indicative that our approach provides a more reusable code against the standard manner. Similar results can be observed in Table 2 for the ProdCons system. (This answer the research question RQ4.)

5 CONCLUSIONS

In this paper, we have presented a novel concurrency programming style for Java programs, known as Aspect-Oriented JCSP - AJCSP. This new style uses JCSP features to add concurrency behavior. JCSP is a framework that implements CSP features in Java language. By using JCSP one can abstract the use of Java threads, whereas the main reason to use AJCSP is to abstract the use of JCSP framework as a whole. With AJCSP, a programmer writes special annotations in the sequential Java code. Such annotations are a CSP-like syntax. We use a compiler that translates AJCSP annotations into AspectJ aspects with JCSP code. Such aspects are responsible for adding the concurrency behavior in a implicitly way. Currently, we are also conducting more case studies to evaluate qualities and limitations of AJCSP.

We believe that the usage of aspects to implement concurrency concern with JCSP introduces a new level of modularity. In other words, our approach is not invasive (the Java source code is not tangled and scattered with the JCSP concurrency code). This gives more flexibility to maintain the source code. To better explain the impacts of AJCSP approach, we have conducted a case study on two Java programs. We implemented those systems in three different ways: AJCSP, JCSP, and Java threads. We used metrics such as separation of concern, coupling, and size to evaluate our claims about modularity in concurrent JCSP programs. The results provided evidences that AJCSP may improve modularity of concurrent systems. Eventually, due to the simplicity of our approach, we can argue that the maintenance effort is minimized when using AJCSP to develop concurrent programs.

REFERENCES

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE TSE*, 20:476–493.

Eaddy, M. et al. (2008). Do crosscutting concerns cause defects? *IEEE TSE*, 34(4):497–515.

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D. (2006). *Java Concurrency in Practice*. Addison-Wesley, Upper Saddle River, NJ.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA.

Roscoe, A. W., Hoare, C. A. R., and Bird, R. (1997). *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Sant'anna, C. et al. (2003). On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings of SBES'03*, pages 19–34.

Welch, P. (2006). Jcsp: Communicating sequential processes for java. http://www.cs.kent.ac.uk/projects/ofa/jcsp/.