An Annotation-Based Approach for JCSP Concurrent Programming: A Quantitative Study

José Elias Araújo[1], Henrique Rebêlo[1], Ricardo Lima[1],
Alexandre Mota[1], Uirá Kulesza[2], Cláudio Sant'Anna[3]

[1] Informatics Center, Federal University of Pernambuco
50740-540, Recife, PE, Brazil

[2] Department of Informatics and Applied Mathematics, Federal University of Rio Grande do Norte
59072-970, Natal, RN, Brazil

[3] Departament of Computer Science, Federel University of Bahia
Salvador, Bahia, Brazil

{jeqca,hemr,rmfl,acm}@cin.ufpe.br, uira@dimap.ufrn.br, santanna@dcc.ufba.br

ABSTRACT

The construction of large scale parallel and concurrent applications is one of the greatest challenges faced by software engineers nowadays. Modern programming models for concurrency including libraries implementing high level abstractions such as JCSP lead to tangled and scattered concurrency code. As such, this paper outlines our initial effort on the separate of concurrent (JCSP code) concern from the sequential Java processes. We explore metadata annotations to implement this separation of concerns. A compiler generates AspectJ code used to instrument the JCSP features under the hood. We also present a case study that assesses the benefits of the proposed approach through a metrics suite.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Aspect-Oriented Programming*; D.3.2 [**Programming Languages**]: Languages Classifications—*CSP*

General Terms

Measurement, Design, Experimentation

Keywords

Concurrent programming, CSP language, metadata annotations, aspect-oriented programming, AspectJ

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) [9] 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* [3]. 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 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 three 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 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 quantita-

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

Figure 1: Producer and consumer in CSP.

tively 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 [9] 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 [7, 4] 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 [4], 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.

Figure 2 presents the CSP (Figure 1) counterpart in JCSP. As observed, JCSP processes are plain Java classes which implement the interface `org.jcsp.lang.CSProcess`, where the method `run` is mandatory. The channel field `ch` (line 3) is used for outputting values (this is denoted by the keyword `ChannelOutput`). The method `produce` captures the CSP construct `|~| data: 1..100 @` as a function which returns a random value from 1 to 100. As mentioned, the method `run` is mandatory. It represents the concurrent behavior of a particular class. In this case, it 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 22) class implements the interface `org.jcsp.lang.CSProcess` and provides a method `run` to be implemented. The `ch` attribute is used to input values. Thus, this process starting reading a value available in the `ch` channel and store it in another field named `data`. The method `print` captures the CSP construct `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`. It captures the parallelism between processes `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 [9]. 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 conventional 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`.

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

```
public class Program {
  public void main(){
    One2OneChannel ch;
    ch = Channel.one2one();
    Producer prod;
    prod = new Producer(ch.out());
    Consumer cons;
    cons = new Consumer(ch.in());
    Parallel parallel = new Parallel();
    parallel.addProcess(prod);
    parallel.addProcess(cons);
    parallel.run();
  }
}
```

Figure 2: Producer and Consumer classes implemented with JCSP.

```
1  //@# var data
2  //@# data = produce() -> ch!data -> Producer
3  public class Producer {
4   public Object produce(){...}
5  }
6  //@# var data
7  //@# ch?data -> print(data) -> Consumer
8  public class Consumer {
9   public void print(Object data){...}
10 }
11 //@# Producer[||] Consumer
12 public class Program {
13  public static void main(String[] args){
14   (new Program()).run();
15  }
16 }
```

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

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, we 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.

4.1.1 *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.

4.1.2 *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?

4.1.3 *Metrics Suite*

In order to answer the research questions, we selected a metrics suite proposed in [8, 2, 6] to evaluate separation of concerns, coupling, and code size. These metrics were adapted form classic OO metrics [1] to be applied to the AOP paradigm.

Lower values for a given metrics implies better results, for instance the two versions of scattering metrics (DOSC and DOSM) varies from 0 (completely localized) to 1 (completely delocalized, present in all components). The degree of scattering (DOS) metrics (DOSC and DOSM) metric created by Eaddy, Aho, and Murphy [6] not only considers which elements are involved in the implementation of a concern, but also how the code is distributed among those elements.

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 SoC, CBC, and size metrics, refer to [1, 8].

4.1.4 *Assessment Procedures*

We implemented three versions (using AJCSP, JCSP, and Java threads) of three different applications: Producer and Consumer (ProdCons), bingo game (Bingo), and Alarm-

Table 1: Separation of Concerns Metrics Results.

System	Version	CDC	CDO	LOCC	DOSC	DOSM	DOTC
ProdCons	Threads	4	8	55	0.86	0.92	0.66
	JCSP	3	6	44	0.99	0.93	0.41
	AJCSP	3	0	6	1	0	0.8
Bingo	Threads	4	10	21	0.95	0.88	0.42
	JCSP	3	7	50	0.97	0.85	0.61
	AJCSP	3	0	6	0.97	0	0.21
AlarmClock	Threads	4	5	36	0.85	0.83	0.59
	JCSP	4	4	64	0.86	0.75	0.65
	AJCSP	4	0	8	0.89	0	0.27

Clock [5]. We implemented 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) [8, 6] 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.

The DOSC metric is used to 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. In other words, by using AJCSP, we still have the scattering effect of the crosscutting concern concurrency. Even though, the AJCSP presents the worst results related to DOSC, we can see that the other two versions presented similar high degree of scattering across components (e.g., classes). As discussed, to complement the DOSC metric, and to know in how many components the concurrency concern is scattered, we decided to also employ the CDC metric. This way, the measure of CDC presented similar results for the three analyzed 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 total separation of concern regarding scattering across classes.

Since all versions (by using both DOSC and CDC metrics) demonstrated to be too scattered in relation to their components, we decided to analyze the scattering in a more fine grained way. Thus, we employed the DOSM metric to 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 DOSM. However, since the two counterparts still have similar DOSM, we also employed the CDO metric to complement DOSM. Hence, we can observe that the JCSP approach is better, against Java threads, due to less operations to change in a eventual evolution scenario. Therefore, we can conclude that AJCSP is a superior solution for both CDO and DOSM metrics. 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 (mainly DOSM and CDO) for scattering measurement, the AJCSP approach is the answer for the first research question (RQ1). The fined grained scattering metrics (DOSM and CDO) were essential to find the real benefits of AJCSP in relation to its counterparts in JCSP and threads.

Code tangling is another symptom of non-modularization of a particular concern. 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 to the other approaches (JCSP and threads) in the Bingo and AlarmClock systems.

The measure of lines of code (LOC metric in Table 2) of the ProdCons system is reduced in 70% in relation to the other ones (which are 27% and 23% for Bingo and AlarmClock, respectively). Since the tangling degree is *inversely proportional* to a system size, and since the LOC of the ProdCons was quite reduced; this justifies the higher tangling of the AJCSP version presented in the ProdCons system. 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). Proportionally, we can argue that our approach is less tangling since the use of AJCSP reduces too much the LOC of the application.

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

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

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
AlarmClock	Threads	177	11	21	6	8
	JCSP	200	16	24	6	15
	AJCSP	154	9	22	4	3

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 refactored 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 for the AJCSP version showed to be 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 and AlarmClock systems. (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 later 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 and AlarmClock systems. (This answer the research question RQ4.)

5. CONCLUDING REMARKS

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.

As future work, we intend to use programming laws to formalize the translation strategy and to ensure the soundness of our approach. Currently, we are also conducting more case studies, with different sizes and complexity, to evaluate qualitatively and quantitatively the AJCSP approach. Also, with an in-depth study, we can investigate in more details the drawbacks that our approach can have.

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 three Java programs. One of them extracted from an open source repository [5]. 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 approach.

6. REFERENCES

[1] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20:476–493, June 1994.

[2] A. Garcia et al. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proceedings of the 4th AOSD (AOSD'05)*, March 2005.

[3] B. Goetz et al. *Java Concurrency in Practice.* Addison-Wesley, Upper Saddle River, NJ, 2006.

[4] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[5] S. Khurshid et al. Software-artifact infrastructure repository. http://sir.unl.edu/content/sir.html.

[6] M. E. others. Do crosscutting concerns cause defects? *IEEE TSE*, 2008.

[7] A. W. Roscoe et al. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[8] C. Sant'anna et al. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings XVII SBES.*

[9] P. Welch. Jcsp: Communicating sequential processes for java., February 2006. http://www.cs.kent.ac.uk/projects/ofa/jcsp/.

A. Online Appendix

We invite researchers to replicate our study. Source code for the subject programs and their implementation versions in both JCSP and AJCSP, our AJCSP compiler, and our results are available at `http://cin.ufpe.br/~jeqca/miss11`.