Adding Autonomy into Object

Jamil Ahmed[1] **and Sheng Yu**[1]
[1]Department of Computer Science, University of Western Ontario, London, Ontario, Canada

Abstract - *Real world objects can be classified into two kinds according to their behavior (1)autonomous objects (2)dependent objects. An object can behave both ways as well. Dependent objects are those objects which are of no use unless exploited by an external entity. Once they are created or instantiated, they keep waiting for the driver class to invoke theirs functions for their utilization. Example of dependent objects include a car, a calculator, a word processing application etc. Autonomous Objects are those objects which when created or instantiate, then they know by their self what they are supposed to do and then they readily start performing their task (set of methods) with possibly no external interaction or invocation. We emphasize that autonomy of object intuitively needs to have these two properties (1)Object runs its method(s) itself as soon as it is created. (2)More than one copy of object can be running simultaneously. Example of autonomous objects include a clock, a car set at cruise control, an Operating system kernel that always keeps active, a virus scan utility that always keeps active, Graphical Actors(simulation of humans) in game programming, an automatic robot, etc. We have established object calculus of autonomous object definition & object creation which incorporates the intuitive properties of autonomous objects as well. Our proposed calculus is based on the same structures as that of Abadi & Cardelli [1].*

Keywords: Object Oriented Programming, Autonomy, Concurrency, Multithreading, Object Calculus.

1 Introduction

Contemporary object oriented programming languages do not yet explicitly provide any feature of "autonomy" for objects. We propose that an object in Object Oriented Programming can be defined as autonomous object. Adding the feature of autonomy to object has its own intuitive effects and introduces new abstraction to some programming languages contemporary features, while reducing the complexity of those features. Autonomous objects provide much more intuitive mechanism of programming for any autonomous computing e.g autonomous vehicles and robots, Graphical Actors(simulation of humans) in game programming, a virus scan utility that remains active all the time etc. In order to mechanize autonomous behavior as natural and intuitive, we introduce two components: (1) A compulsory run() method that will get invoked by default to start the function of autonomous object as soon as object is created. (2)When an autonomous object is created, it is by default created in its own separate and new thread. All our code examples in Figures are analogous to 'java' syntax. Autonomy of objects also provide an abstraction to an important contemporary programming language feature, making Object oriented programming closer to natural way of programming and hiding much of the complexities of that language features. We propose that following feature gets new abstraction by virtue of the notion of "autonomy".

- Concurrency (Multithreading)

As we argue that Autonomous object provide concurrency which is more intuitive and close to the concurrency of real world objects because with autonomous object we do not need to explicitly care about threads just like real world objects.

1.1 Autonomous Object Definition

We introduce a special method "run()" as a mandatory method of object definition for the object which is supposed to behave like autonomous object. This method is supposed to be invoked by constructor method of autonomous object by default. The purpose of "run()" method is to define in its body that what operations this autonomous object must start performing right after its creation. We give structure of autonomous object definition code below.

```
Class auto_class{
        …..
        Public void run()
        {….. }
        …..
        public   auto_class ()
        {…..}//constructor is optional as
        usual
        …..
}
```

Fig. (A)

1.2 Autonomous Object Creation

We introduce the keyword "auto" to be used in conjunction with autonomous object creation. This "auto" keyword will force the object created as autonomous objects. Whenever an object is created with "auto" keyword, the compiler expects that the object must have special method "run()" defined in its definition. An attempt to create autonomous object using "auto" keyword will generate compile time error if the object definition doesn't have "run()" method defined. We give autonomous object creation code below.

This research is supported by Higher Education Commission, Pakistan (www.hec.gov.pk) and NSERC (Natural Science and Engineering Research Council of Canada) grant number 41630.

```
Class Driver
{
  main(){
  …..
auto_class   auto   objA =new  auto_class ();
…..       }
}
```

Fig (B)

If the object definition have "run()" method defined but the object is not created with "auto" keyword then there will be no compiler error. Object creation without "auto" keyword will lead to usual object i.e non-autonomous or dependent object creation. The object created without "auto" keyword will not invoke the "run()" method, if there is any.

2 Concurrency by Virtue of Autonomy

Although the feature of concurrency is already provided by contemporary programming languages but this feature is provided by introducing additional and distinct entity rather than a built-in feature of object which makes it language based feature rather than a built counterpart of object. Those distinct entity (e.g thread) are then applied on objects and this is how objects can exploit the feature of concurrency so far. Although concurrency with the help of distinct entity like thread also gives a certain level of autonomy to Objects but as we propose autonomy as a built in feature of objects which cause concurrency to be a rather naturally associated and inherent to autonomous objects. Consequently the concurrency feature of programming languages will be under the hood of autonomous objects. Once we have autonomous objects, these independent autonomous objects will be well suited to inherently have concurrency capabilities.

Hence , notion of autonomous object provide new abstraction to thread such that each new instance of autonomous object will be created in its new and separate thread. This notion helps us to get rid of explicitly thinking in terms of thread and creating threads on our own. All we need to think about is Autonomous object. We won't need to know new language based ways (e.g thread libraries) and constructs or syntax for implementing and exploiting concurrency as we do by now in contemporary programming languages. With autonomous object, creating new thread and its handling won't be the responsibility of programmer any more.

This new abstraction will also be conducive in hiding many contemporary issues of multithreading (explicitly creating and destroying thread, races between the threads, deadlocks etc). It means multithreading will then become an inherent part of autonomous objects. All the principles and practices of concurrent programming (such as races between the threads, locks, deadlocks etc) will remain intact. The only differences will come up is that the thread management will be taken care of by autonomous object. Hence, the difference will appear in the view point by achieving higher abstraction to concurrent programming.

When an autonomous object is created, it is by default created in its own separate and new thread. The keyword "auto" instructs the compiler that the objects created will run in separate thread. Hence, this reserve word causes the compiler to create a new thread by default so that this object is run on top of that thread.

The object created without auto keyword will not invoke the "run()" method and a separate thread will not be created.

3 Architecture of Autonomous Object and Concurrency

We propose architecture of Autonomous object and its concurrency in Fig (C). We introduce a new built in class of object oriented system in the compiler called "Autonomous" class. "Autonomous" class works in collaboration with built in "thread" class. Whenever an object is created with "auto" keyword, it gets inherited by the built in "Autonomous" class by default. The "auto" keyword invokes its "run()" method from within its constructor. It also enforces to inherit the autonomous object from a special class "Autonomous". This "Autonomous" class in turn create a new thread object within it, using "Has a" inheritance, on top of which a newly created autonomous object will run. The "run()" method of autonomous object will by default invoke the "super.run()" instruction to override the "run()" method of "Autonomous" class.

Each autonomous object, when created, by default creates a "thread" object internally and hides thread level details inside it, thus providing a single and higher abstract level of concurrency. In other words we can say each autonomous object is created on top of a thread object to ensure autonomy. This thread is called primary thread of the object. Figure (C) gives our porposed architecture for any object definition say "auto_class" when an object of this class is created as an autonomous object.

To exploit autonomous object we show a driver class with "main()" function in which we can create one or more object as autonomous object. Each of those objects when created will get functional in separate threads synchronously. Compiler will take care of thread creation responsibility. The "driver" class and "auto_class" are user defined whereas the "autonomous" class and "thread" class are built in class which gets associated with the "auto_class" by the compiler.

By virtue of autonomy we have introduced a new abstraction for multithreading. Now we can exploit more than one autonomous objects to perform their operation concurrently. When two instances of same autonomous object are created, it is equivalent to two threads performing an operation concurrently.

In section 6 we show examples of contemporary code of multithreading (Example2-code#1) and equivalent code proposed by us (Example2-code#2) according to the notion of concurrency by autonomous objects.

```
Class Thread
{ ….
        run(){….}
        Synchronized(Object obj){….}     //this
        //method will lock the object's methods to
        //be called by two threads at the same time
        isAlive(){….}    //determines whether a
        //thread is still running
        Join(){….}//causes the main thread //(from
        where it is invoked) to  wait until //the child
        thread terminates  and "joins"  main thread.
        ….
}
Class Autonomous
{       …
        Public Thread  thread;
        Autonomous()
        {   thread=new Thread(); }
        …
        run()
        {    thread.run();       ….   }
        …
}
Class auto_class
{       …
        run()  // super.run() is invoked by default
        { ….}
        auto_class () // constructor
        { …..    }
        …
}
Class driver
{       …
        auto_class    auto   objA=new auto_class ();
        //creating objA with "auto" keyword will
        //force "auto_class" to get implicitly
        //inherited from "Autonomous" class.
        ….
 }
```

Fig (C)

4 Motivation of Concurrency by Autonomous Objects

Autonomous Objects will be the primary and base entity for concurrency instead of thread. Autonomous object notion provides higher abstraction to the widely varying language constructs and libraries of thread. Since concurrency is now represented by Objects at the higher abstract level, it will be very intuitive to define the soundness of Object calculi for autonomous object with built in concurrency feature. We won't need to introduce a separate entity, within the calculus, to represent threads unlike most of the alternate object calculus which introduces new constructs within calculus to incorporate thread and concurrency as in [3] , [4].

Objects, as an abstract entity for concurrency, represent more natural point of view for multithreading much closer to the concurrency point of view of real world objects as illustrated ahead by some multithreading scenario.

5 Single Threaded Autonomous Object

Example 1 illustrates a single threaded autonomous object. Main thread of driver class creates only one object with "auto" keyword to get only one new thread for autonomous object.

5.1 Example 1

In this example we have given object definition of an Autonomous "AutoPrinter" object. As soon as an autonomous object is created, the autonomous printer object is supposed to start printing autonomously without any external request. Within run() method we have defined the startprinting() method. When object is created using "auto" keyword then AutoPrinter object gets implicitly inherited by built in class "Autonomous", a new thread is created on top of which this object gets functional and "run()" method is invoked, by default, implicitly and synchronously from within the constructor of AutoPrinter. The implicit call to run() is taken care of by the compiler.

Example 1

```
class AutoPrinter{
  String name;
  int i=0;
  AutoPrinter () {
        System.out.println("Auto. Thread started ");
   }
   public void run() {
           StartPrinting();
   }
   Public StartPrinting(){
       While(i<100){
        System.out.println("Print in progress");
        }
   }
}
class Driver{
   public static void main (String args[]){
      System.out.println("Main thread started");
      AutoPrinter auto objA =new AutoPrinter ();
      System.out.println("Main thread terminated");
   }
}
```

Fig (D)

6 Synchronization

While exploiting multithreading, there are times, when more than one thread share the same resource. More than one thread can invoke the same method of that resource at the same time. Obviously only one of them should be allowed to access the resource/method at one time. In this case we need to synchronize the threads.

Example2, code#2 illustrates a multithreaded autonomous object as two object are created with “auto” keyword. In this Example, we show by comparison that how autonomous object serves perfectly well in code#2 as an alternate of contemporary multithreading technique in code#1. Code#2 hides all thread level management code so that we can best appreciate the simplicity of code#2 and realize the abstraction of “concurrency by virtue of Autonomy”.

6.1 Example 2

Example 2—Code#1

```
class Printer  {
   void Printlist (String s) {
     System.out.print ("printing long list for”+s);
   try {
      Thread.sleep (1000);
   } catch (InterruptedException e) {
        System.out.println ("Interrupted");
   }
     System.out.print ("printing long list Ends for”+s);
  }
}
class CompThread implements Runnable {
   String s1;
   Printer p1;
   Thread t;
   public CompThread (Printer p2, String s2) {
      p1= p2;
      s1= s2;
      t = new Thread(this);
      t.start();
   }
   public void run() {
       synchronized(p1){
          p1.Printlist(s1);
       }
   }
}
class Driver{
   public static void main (String args[]) {
     Printer p3 = new Printer();
     CompThread name1 = new CompThread (p3, "Bob");
     CompThread name2 = new CompThread (p3,"Mary");
     try {
        name1.t.join();
        name2.t.join();
     } catch (InterruptedException e ) {
          System.out.println( "Interrupted");
     }
  }
}
```

Fig (E)

Fig (E) gives a scenario using contemporary techniques of concurrency in java. The method “Printlist” of printer is shared by two computer threads. We have explicitly synchronized the call to this method so that both threads do not intermingle their execution of this method. Synchronization statement clocks the invocation of this method by other thread as long as the execution of this method by first thread is under process.

Fig (F) gives alternate solution of this problem by exploiting the inherent power of autonomous object of our proposal. In code#1 we have to explicitly care about creating threads and using shared resource within the threads and then synchronizing. Where as in code #2 we can simply define the shared printer as an autonomous object and every time a new autonomous object is created, compiler will itself take care of the new thread creation issues.

This example best realizes the significance of abstraction provided by the notion of Autonomy. We can see that we gain same multithreading in code#2 as in code#1 but we don’t even need to explicitly think about thread creation in code#2. All thread creation, for the sake of implementation, is done internally under the abstraction of autonomous object.

Example 2—Code#2

```
class AutoPrinter  {
             public string pname;
             public void run() {
                      thread.synchronized(){
                      this.Printlist(pname);
                    }
                 }
             public AutoPrinter (string nm){    pname=nm;      }
             public void Printlist (String s) {
                  system.out.print ("printing long list”+s);
                  try {
                     Thread.sleep (1000);
                  } catch (InterruptedException e) {
                     System.out.println ("Interrupted");
                 }
                  System.out.print ("printing long list Ends”);
             }
}
class Demo{
              public static void main (String args[]) {
                try {
                AutoPrinter auto p1 =new AutoPrinter("Bob");
                AutoPrinter auto p2 =new AutoPrinter("Mary");
                }   catch (InterruptedException e ) {
                System.out.println( "Interrupted");
                        }
               }
}
```

Fig (F)

7 Calculus

A class is an object definition used to generate object. Pre-methods are the method definitions which becomes methods once embedded into objects as mentioned in [1]. A class is a collection of pre-methods together with a method called "*new*" for generating new objects. Class in the terminology of calculus is written as below:

$c \ \underline{\Delta}\ [new = \varsigma(z)[l_i = \varsigma(s)z.\ l_{i(s)}\ ^{i\in 1\ldots n}],$

$l_i = \lambda(s)b_i\ ^{i\in 1\ldots n}\ ]$

The method new $= \varsigma(z)[li = \varsigma(s)z.\ li(s)\ i\in 1\ldots n\ ]$, applies the pre-methods of class to the self of the object, thereby converting the pre-methods into methods.

Given any class "c", the invocation c.new produces an object "o" as below and given in [1].

$o \ \underline{\Delta}\ c.new = [l_i = \varsigma(x_i)b_i\ ^{i\in 1\ldots n}]$

7.1 Calculus for Autonomous Object

In our setting, as we have defined in section 3, in order to make a class behave as autonomous, it must be inherit from a parent "Autonomous" class. In our calculus we call it "c_super_auto" class and formally given as below

c_super_auto $\underline{\Delta}$ $[new = \varsigma(z)[l_i = \varsigma(s)z.l_{i(s)}\ ^{i\in 1\ldots n},$
$\varsigma(s)z.run(s)]$, thread=b,
run=$\lambda(s)$(s.thread:=s.thread.new),
$l_i = \lambda(s)b_i\ ^{i\in 1\ldots n}\ ]$

- thread=b stands for thread= $\varsigma(s)$b, for an unused s because thread=b is a field.
- run=$\lambda(s)$(s.thread:=s.thread.new), A new instance of thread is created, so that each autonomous object can run in this new and separate thread.
- (s.thread:=s.thread.new) is the body of *run* method.
- *new* method not only applies the pre-methods of class to the self of the object but also invoke the *run* method.
- *run* is also a special method similar to new method. A new thread instance is created within *run* method. Hence a new thread instance is created before new method is returned.
- $l_i = \lambda(s)b_i\ ^{i\in 1\ldots n}$ represents all pre-methods of "c_super_auto" class.

In order to make a class behave as autonomous, it must be inherit from a parent "c_super_auto" class.
As soon as object of a class is created with "auto" keyword, the class by default gets inherited from "c_super_auto" class. Compiler is supposed to enforce this by default inheritance. In our calculus, we call the inherited class "c_auto" and formally given as below:

c_ auto $\underline{\Delta}$ $[new = \varsigma(z)[l_i = \varsigma(s)z.\ l_{i(s)}\ ^{i\in 1\ldots n+m},$
$\varsigma(s)z$ .run(s)],
run= $\lambda(s)$c_super_auto.run(b_r)(s),

l_i =c_super_auto.$l_j\ ^{j\in 1\ldots n}$,
$l_k = \lambda(s)b_k\ ^{k\in n+1\ldots n+m}\ ]$

- "c_auto" as an inherited class can reuse all the pre-methods of "c_super_auto".
- (c_super_auto.$l_j\ ^{j\in 1\ldots n}$) are the pre-methods of "c_super_auto" inherited into "c_auto".
- $l_k = \lambda(s)b_k\ ^{k\in n+1\ldots n+m}$ are more pre-methods peculiar to "c_auto".
- run= $\lambda(s)$ c_super_auto.run(b_r) (s) shows that *run* is the inherited but over ridden pre-method which also invoke its parent's *run* pre-method.
- c_super_auto.run(b_r) (s) is the body of *run* pre-method of "c_auto".
- c_super_auto.run is the part of *run* pre-method body which is invoking the *run* method of parent.
- (b_r) is the remaining body of *run* pre-method which represent any custom user defined code.
- (s) is the self parameter of "c_auto" also passed on to c_super_auto.run

In our calculus an autonomous object "ao" is created from "c_auto" class is formally given as below:

ao $\underline{\Delta}$ c_auto.new = [b_r {{x<-ao}}, $l_i = \varsigma(x_i)b_i\ ^{i\in 1\ldots n+m}$]

- b_r {{x<-ao}} shows that *run* method is invoked from within the c_auto.new i.e as soon as "ao" is created.
- $l_i = \varsigma(x_i)b_i\ ^{i\in 1\ldots n+m}$ shows the methods embedded into "ao" corresponds to the pre-methods of "c_auto" class. These methods include the 'm' pre-methods of c_ auto as well as 'n' pre-methods of c_super_auto.

8 Conclusion

Our proposed autonomous object notion is compatible with all contemporary Object Oriented Programming techniques. It is not new programming paradigm. According to our proposed syntax when an Autonomous object is defined, its object can still be created as a usual, as a non-autonomous object, without any extra care. Autonomous object provides better abstraction over thread and concurrency and is also sound in Object calculus as we have shown.

9 References

[1] M.Abadi and L.Cardeli, “A Theory of Objects”, Springer, New York, 1996. Chapter 6.

[2] Michael Papathomas, Dimitri Konstantas, “Integration concurrency and Object Oriented Programming. An Evaluation of Hybrid”

[3] Andrew D. Gordon, Paul D. Hankin, University of Cambridge Computer Laboratory, Cambridge, UK, “A Concurrent Object Calculus: Reduction and Typing”.

[4] Alan Jeffrey, DePaul University, “A Distributed object calculus”, December 1999, Proc. FOOL 2000

[5] Suresh Jagannathan!, Jan Vitek, Adam Welc, Antony Hosking, April 2005, Dept. of Comp. Sci., Purdue University, USA “A transactional object calculus”.

[6] Jonathan Aldrich Joshua Sunshine Darpan Saini Zachary Sparks, School of Comp. Sci., Carnegie Mellon University, OOPSLA 2009, “Typestate-Oriented Programming”.

[7] Haitong Wu, Sheng Yu, Dept. of Comp. Sci, Univ. of Western Ontario, “Adding states into Object Types”.

[8] Haitong Wu, Sheng Yu, Dept. of Comp. Sci, Univ. of Western Ontario, 2006 Elsevier, “Type Theory and Language Constructs for Objects with States”.