Enhancing Modularity in Aspect-Oriented Software Systems-An Assessment Study

Kotrappa Sirbi
Department of Computer Science & Engineering
K L E's College of Engineering & Technology
Belgaum, India
kotrappa06@gmail.com

Prakash Jayanth Kulkarni
Department of Computer Science & Engineering
Walchand College of Engineering
Sangli, India
pjk_walchand@rediffmail.com

***Abstract*—Aspect-oriented programming (AOP) is rapidly gaining popularity among research and industry as a methodology that complements and extends the object-oriented paradigm.AOP promises to localize the concerns that inherently crosscut the primary structural decomposition of a software system. Localization of concerns is critical to parallel development, maintainability, modular reasoning and program understanding. However, AOP as it stands today is bringing problems in exactly these areas, defeating its purpose. Previous work and experience gleaned from building AOP systems have identified two points of contention that are impeding the adoption of AOP. First, the complication arising from the need to open up systems modules for AOP and the need to protect those modules against possible fault injection by AOP. Second, the need to have base system components stabilized before aspect components can be developed. Clearly, this adversely affects parallel development. This dependency also causes aspect components to be sensitive to changes in the base system, complicating maintainability, already a high-cost element in the software process. In this review paper, we argue that the AOP provides better modularity and here main focus is to assess the solution that affords better modularity to AOP systems.**

Keywords-Aspect Oriented Software Development (AOSD), Aspect Oriented Programming (AOP), Aspect Oriented (AO) System, AspectJ.

I. INTRODUCTION

Jacobson [1, 2] believes that aspect-oriented software development (AOSD) will let us gracefully extend an existing software system iteration-by-iteration and release-by-release for the system's entire lifetime. AOSD [7, 9, 10] will have an impact on most software measures including, cost, quality, and time-to-market. Costs can see a reduction of at least 20% [1, 2]. This view may find support in the growing array of application areas where aspects are being deployed. So far, AOP has reached logging, tracing, profiling, buffering, pooling, caching, synchronization, error handling, load balancing, persistence, mobile applications code base retargeting, schema versioning in databases, security, and transaction management.

AOP promises to localize crosscutting concerns by providing language-based mechanisms for explicitly representing their structure and/or behavior. To fulfill its promise, AOP relies on the ability to impact a large body of the base code at once from the aspect code side. We believe that AOP does provide a cleaner separation of concerns. However, we also believe that AOP has slowly moving towards maturity when it comes to the benefits of better modularity. This view is shared by others in the AOP community [3].It is important to note that AOP is not a silver bullet for separating all crosscutting concerns. In fact, all crosscutting concerns should not be implemented as aspects and should not be separated from the base code. An example is given [4] for the concerns of concurrency and failures. The authors present an example where separation of concerns into aspects led to an irresolvable deadlock. Applying AOP mandates an understanding of its limitations by system designers/developers.

II. BACKGROUND

A. Open Modules

The Open Modules system with the root idea that base code and AOP code should be protected from one another. This approach is a shift from pure obliviousness. In Open Modules, a module exposes pointcuts that can be advised by external aspects as part of the module's interface. OpenModules includes a module system that allows external aspects to advise the exported interface but prohibits external aspects from advising calls originating from within the module. The aspect impact is limited only to calls originating external to the interface, which provides decoupling of aspect code from the implementation details of the code it applies to. Open Modules exposes the module's entry joinpoints that can be safely advised. The semantics allow private state to be advised if the module declares a pointcut allowing quantification to include private state. Otherwise, private state is not advisable.

This solution seeks to strike a balance between openness and modularity. However, in Open Modules, the advice does not crosscut modules that are non-hierarchically related. Hence, not all aspects can be implemented easily and there is no guarantee that existing aspects would not have to change as the base code incorporates new functionality.While Open Modules provides better modularity, using Open Modules approach for the RideArrival system[17] would require duplication of persistence code for components and it also require duplication of logging code for components . This is because Open Modules does not allow advice to crosscut modules. In addition, this restriction complicates future

functional extensions that are crosscutting in nature, we are not sure if they are even possible to implement as cleanly isolated aspects.

B. Aspect-Aware Interfaces

Aspect-Aware Interfaces (AAI) were developed by Kiczales and Mezini [14] to demonstrate that AOP provides support for modular reasoning of crosscutting concerns in the presence of aspects. Similar to Open Modules, AAI move away from pure obliviousness. Consider the code snippet of figure1 shown on the following page.

The code is borrowed with minor modifications from [14].The method moveby(int, int) has three new pieces of information as part of its interface: the name of the aspect that may advise the method, the kind of the advice (after returning), and the name of the pointcut to which the advice is attached in the aspect code. The aspect's interface includes the inverse information about which methods it affects.

The base code interface is now "aware" of two aspect-related pieces of information:

1. Aspect UpdateSignaling has an after returning advice that affects the execution of method moveby(int, int) of the interface.

```
class Point implements Shape {
void moveby(int deltax, int deltay):
UpdateSignaling - after returning
UpdateSignaling.move();
}
aspect UpdateSignaling {
after returning: UpdateSignaling.move():
Point.setX (int), Point.setY (int),
Point.moveby (int, int);
}
```

Figure 1: Aspect-Aware Interfaces

2. Aspect UpdateSignaling has a pointcut move() that selects executions of method moveby(int, int) of class Point as joinpoints.Now, the base code developer is aware of aspect advising, the aspects involved are known, and the expected aspect effect is also known (what type of advice). While this is better than programming base systems purely oblivious of aspects, especially for reasoning purposes, this design limits the effectiveness of modular development for the following reasons:

i. The aspect is sensitive to base code interface changes because the pointcuts are coded using base code names directly. The aspect code has to wait for the interface to be stabilized before aspect development can start.

ii. The aspect-related information are computed and attached to methods, but are not part of the code. Unless comments are included with a class, nothing indicates that it is being advised.

These limitations, especially (1), made it difficult to adopt AAI's approach to implement crosscutting concerns in the RideArrival system [17].

AAI extend the interfaces of base code modules through advising as opposed to extending an interface through implementation or interface inheritance. AAI recognize that the complete specification of a module's interface in an AOP system depends on the complete system configuration. This means that a module's interface is dependent on the interfaces referenced by that interface and the interfaces of aspects advising it. The "self-contained" property of modules ceases to hold because of the mutual contribution to the interfaces of base code and aspects by one another. AAI also recognize the need to program against crosscutting interfaces, similar to Crosscutting Interfaces discussed shortly. However, they are not geared towards loose coupling of aspects and do not provide control and protection over aspect and base code interaction, unlike Crosscutting Interfaces.

C. Crosscutting Pointcut Interfaces

Crosscutting Pointcut Interfaces (XPI's) [11] attempt to decouple the aspect code from the base code. The idea is to dissect a traditional aspect into three aspects that cooperate to realize a crosscutting concern. An aspect is used for specifying pointcuts and advising constraints. Another aspect is used for advice implementations, and a third aspect for checking advising constraints. XPI's provide better decoupling of advice implementation from base code changes. An advice implementation aspect is coded in terms of the exposed pointcuts of another aspect instead of depending on the base code directly. Through checking of advising constraints, XPI's also provide some level of protection to the base code, even though the aspect side controls it. The base code is still passive in this regard. This is problematic for the RideArrival system [17] implementation of persistence because it requires the base code components to fully trust the aspect implementation, which does not guarantee that harmful aspects are kept at bay.

D. Explicit Joinpoints

Explicit joinpoints (EJP's) are a language mechanism devised by [13] that also shares AOP community view of a shift away from pure obliviousness. It is worth noting that this work was developed independently from EJP. The base code in this system has to be prepared by inserting syntactic hooks that look like static method calls at points where advising is required. Explicit joinpoints have scopes attached to them which extend AspectJ's power fundamentally as it allows advising on arbitrary blocks of code. EJP's is a more robust solution than AspectJ because advising is not unilateral anymore; the base code marks exactly the points of advice application. The joinpoints are declared in a separate aspect and then referenced by both, the aspect implementing a concern and the base code expecting the concern service. This mechanism avoids the need to refactor the base code for the sake of exposing joinpoints or simplifying pointcut expressions. EJP's modify pointcut expressions to accommodate matching on the scoped explicit joinpoints' names instead of the base code names directly. This feature reduces the effects of the fragile pointcut problem. As in XPI's and EJP's, we maintain that decoupling aspect code from base code is an important requirement for any real-world AOP system. Another interesting feature of EJP's is the handles list; a list of exception types attached to an explicit joinpoint declaration that constitutes a promise to the base code that these exceptions will be handled in some way by the advising aspect.

E. Modular Aspect with Ownership

Modular Aspects with Ownership (MAO) was developed by [6] as an extension to AspectJ 5. MAO's design applies ideas of ownership and effects annotations to improve modular reasoning in the presence of aspects. Ownership systems require programmers to explicitly attach an owner to instantiated objects in the system. In the context of MAO, owners are concern domains that are explicitly declared. The purpose of bringing this idea into AspectJ is to help programmers and tools identify control and heap effects of advice. A control effect is a change in the control flow of the advised code, whether it is base code or another aspect. A heap effect is a change in object state due to field mutation.MAO introduces two new annotations to AspectJ, @surround and @curbing. These annotations can apply to aspects or advice. Annotating an advice, irrespective of the particular kind, with either one, turns the advice into a control-limited advice. This allows the programmer to express that a particular advice will have no control effects (using @surround annotation), or that the only allowed effects are thrown exceptions (using @curbing annotation). Similar meanings apply to aspects using these annotations. MAO uses the aspect annotation @surround to differentiate between spectator (no control effects) and non-spectator aspects. This is important because spectator aspects can be unconstrained in their advice and can be excluded instantly when reasoning about control effects.MAO refers to non-spectator aspects as assistants to highlight the notion of participation from the aspect side in modifying the behavior of a base code component.MAO also uses explicit assistance acceptance, which means that the base code explicitly announces that it can receive advising from certain aspects. MAO proposes an accept declaration for the advised code to use. Two types of assistance exist. One in which an assistant modifies behavior for a component directly (implementation utilities), and another in which an assistant modifies behavior for clients of a module client utilities. This distinction supports modular reasoning about behavior in the presence of assistants. When it comes to heap effects, MAO's ownership and effects system uses annotations like @readonly and @writes to express potential heap effects of the advice application. These annotations help with associating objects (arguments, target, etc.) involved in the advice context with their heap effects if any. MAO also introduces a simple writes pointcut that further refines advice application by allowing programmers to select joinpoints that produce heap effects on a particular concern domain. The concern domain is passed as a parameter to the writes pointcut.MAO addresses a fundamental impediment to the adoption of AOP, lack of modular reasoning in the presence of aspects. MAO supports modular reasoning using a strong type-theoretic foundation

III. AOP-ASPECTJ

AspectJ an extension to Java is as the representative AOP language [12, 16], being the defacto standard of AOP implementation today. A crosscutting concern is a functional requirement that cannot be implemented in an encapsulated fashion. The inherent nature of such functionality is that it has to be delegated to various, possibly unrelated, components in the system. Hence, it crosscuts the primary structural decomposition of a software system. Even though those concerns may have been modeled separately, maintenance is still a challenge because once implemented they dissolve into the code. Crosscutting concerns lead to the following two phenomena in code. Tangling occurs when code implementing a crosscutting concern is mixed with code implementing primary functionality of a component. Scattering occurs when a concern/functionality is assigned to several components where the assigned concern is unrelated to the key concepts the components model in the application domain. Aspect Orientation (AO) started as a mechanism for dealing with implementation issues pertaining to crosscutting concerns. Focusing on detangling functionality and isolating code that used to be scattered. AO provides code locality of the crosscutting concern implementation using an abstraction referred to as an aspect. AO is on its way to becoming more of a methodology and is being integrated into earlier phases of the software process, as early as requirements specification [7].An Aspect is a unit of abstraction that represents a crosscutting concern. Depending on the implementation/context it has been called a slice as in Hyper/J [8,9], an aspect as in AspectJ [12,16] and a theme as in Theme/UML [9].Ideally, all behavior handled by the crosscutting concern should be localized in the aspect(s) representing it. In order to realize the original behavior of the system after eliminating tangling and/or scattering from primary system components, AOP relies on a core mechanism, advice.

AOP inherited the concpect of advice from Common Lisp [5]. AOP can be viewed as the ability to make quantified statements about sets of points in the program where a piece of behavior (an advice) needs to be executed [7, 9, 10]. A pointcut is a quantification mechanism that expresses the selection of sets of program points where an advice would be executed. A point in the program selected by a pointcut is called a joinpoint. Joinpoints include constructor calls, method calls, accessing/mutating a field, etc. In AspectJ, for example, given a method call joinpoint j selected by pointcut p, there are five advising options. These are: before, after, after returning, after throwing, and around advice kinds. At j, a before/after advice will execute prior/post to the execution of the body of method picked by j. Similarly, after returning and after throwing advices run upon returning/throwing an exception from the method at the call joinpoint. An around advice allows wrapping (and possibly replacing altogether) the method at j within the advice code.

IV. MODULARITY UNDER AOP

Ideally, in a modularly designed system, modules are self-contained, loosely coupled, amenable to parallel development, and the system's composition into a specific configuration can be computed and verified in a reasonable amount of time. Modular reasoning has been defined as the ability to anticipate the behavior of a component, say A, based only on A's interface specification and the interfaces of components referenced by A's interface[14].

We believe this view of modular reasoning is flawed in the case of classical AOP (AspectJ-style implementations). Even

in the ideal case, we know from systems engineering that we cannot completely anticipate the behavior of all components in the system without considering the specific configuration/environment where modules are operating. We agree with the work of aspect-aware interfaces by Kiczales and Mezini [14]. In [14], the authors indicate that an OOP program is NOT the same program after AOP advises it. Neither the class nor the aspect is the unit of modularity anymore. It is not enough to examine the interface of a class/aspect to completely anticipate its behavior after composition. Some subtle interactions between the base code and aspect code may only be discovered after a certain configuration has been computed. This understanding drives this work's underlying philosophy, that since base code and aspect code pariticipate in making up a module's interface, then they should explicitly cooperate to preserve the module's boundary without limiting AO capabilities. We see a module boundary extending beyond traditional class or aspect module boundaries with base code modules being responsible for carving out their module boundaries within the system.

A. A Case Study-RideArrival System

The RideArrival system [17] is a location-based service for cell phone users under development at Cream City Bitworks, Inc., Wisconsin. This system used here just for simulation purpose only and it's an incomplete system as per the literature. The system is intended to attract more customers to the bus transit system by eliminating unnecessary waiting time at bus stops. The system uses GPS technology to track bus locations with respect to a set of predefined landmarks. A user registers for the service using their cell phone or mobile device and receives an alert (an SMS message) when the bus is, say, 5 minutes away from the user-specified bus stop. There are four main components in the system:

1. One that handles raw bus location information as it is coming from the GPS unit. This component runs on a mobile device and sends "cleaned up" location information to the server for further processing.
2. A server component that processes location information of buses. A system policy determined that bus location information is sensitive, so access and manipulations are restricted to certain parts of the system.
3. A client that runs on mobile devices of users for signing up and requesting alerts.
4. A server component that handles users' incoming requests.

Aspects were suggested for implementing the crosscutting concerns of logging and persistence. However, an AO solution had to be dropped because aspects could not afford two main things:

i. The system is under development, so interfaces are not fully stable. Changes to the interfaces would trigger more maintenance on the aspect side, more work and more cost. For example, changes to the interface for book-marking bus stops on users' clients induce changes to the persistence aspect. So aspects have to wait until the book-marking code is stable and there is no guarantee that extending functionality in the future will not break the persistence aspect.

ii. Aspects have the potential of compromising the system's integrity because sensitive data is now accessible through channels outside the interfaces of classes holding them and simply because everything in the class is fair game for an aspect.

V. AOP MODULARITY WITH IMAGE INTERFACE (I2) APPROACH

In AO systems, aspects play the role of structure/behavior modifiers of base components [17]. Currently, in non-trivial systems, aspects are tightly coupled to the base code they operate on. This situation makes independent evolution and development of aspects and base code almost impossible. However, AO was primarily motivated and hailed for better modularity. We would expect better modularity to afford better evolution instead we ended up with the phenomenon called, the "aspect evolution paradox". Even simple changes to interfaces in base code trigger updates in all dependent aspects in the system. In order to resolve this paradox, we need a language-level solution that would allow loose coupling between base code and aspects. This solution should not limit aspect capability, should enhance modularity, and should allow independent evolution of aspects and the base code. The Interface Image approach [17] is new approach for better enhanced modularity.

A. What is an Intercafe Image (I2)?

An Interface Image (I2) is a level of indirection through which all advising requests are carried out. An Interface Image provides a mechanism by which a class exposes a set of joinpoints through aliasing base code interface elements. In addition to aliasing, the image incorporates advising constraints per joinpoint that aspects are expected to honor. Aspects are developed against the aliases defined in the interface images of base code classes. Aspects are not allowed to advise classes directly. So only classes that declare images of their interfaces are advisable. This indirection decouples base code development from aspect development. It also creates a layer where control over aspect impact can be implemented without limiting how an aspect can potentially advise a module.

In this design, an I2 provides the following benefits:

1. A class uses the I2 mechanism to exercise control over what is advisable without limiting aspect-oriented capabilities. The class is now an active participant in the advising process since it is up to the class to expose/hide the joinpoints that help realize its functionality without compromising its integrity. For each exposed joinpoint, advising constraints can be attached to disallow unwanted aspect advising. In addition, constraints allow more control on code instrumentation by turning advice execution on/off, which can be of value during testing and debugging. The I2 semantics is that if a class does not provide an image then it becomes unadvisable. The interface image is the gate through which all advising operations can go into a class. This relates to trait shyness: minimizing communication channels. The controlled flow of aspect activity can also help with program understanding and reasoning in the presence of aspects.
2. If new elements are added to the interface of a class, the interface image does not need to change if the new elements are unadvisable. If they are, the image needs to be updated accordingly, but the aspects are left untouched. Thus

preventing the ripple effects of updating aspects whenever a class' interface changes thereby enhancing maintenance. Parallel development can also benefit from this loose coupling. Interface images and core functionality can undergo a relatively short design phase, and then aspect development and core modules' development can proceed independently.
3. The I2 serves as a specification of advisable interface elements for base code and aspect developers alike.

This work studies the interface image approach in the context of classes only.

B. Banking Authorization

This example is adapted from Laddad's, AspectJ in Action [24]. The example was originally developed by Laddad to showcase modularity of an AspectJ solution over a conventional Java solution. The context is an authorization service in a banking system. The example makes calls to the JAAS (Java Authentication and Authorization Service) API. In this example, we assume the set of operations requiring authorization is exactly the same set requiring authentication. Authentication is confirming to the system that a user entity is indeed "who" they claim to be. Authorization is deciding if an authenticated user has enough clearance to perform certain operations or access certain resources. The base code for this example is shown in figures 2 and 3. The code is simple, performing basic operations on bank accounts, including inter-account funds transfer (figure 3).The AspectJ solution uses an abstract aspect, *AbstractAuthAspect*, that provides an implementation of an authorization protocol using the JAAS API.

```
public class AccountSimpleImpl implements Account {
private int accountNumber;
private float balance;
/* constructors and accessors omitted for brevity */
public void credit(float amount) {
balance = balance + amount;
}
public void debit (float amount)
throws InsufficientBalanceException {
if (balance < amount) {
throw new InsufficientBalanceException(
"Total balance not sufficient");
} else {
balance = balance - amount;
}
}
}
```

Figure 2: Class AccountSimpleImpl [Laddad]

This abstract aspect provides an abstract pointcut authOperations() as a hook for concrete derived aspects to quantify which operations in the system they want to apply the authorization protocol to. A derived concrete aspect, *BankingAuthAspect*, that fully implements the authorization concern.

```
package banking;
public class InterAccountTransferSystem {
public static void transfer (Account from, Account to,
float amount)
throws InsufficientBalanceException {
from.debit (amount);
to.credit (amount);
}
}
```

Figure 3: Class InterAccountTransferSystem [Laddad]

Here we included brief about Banking Application, complete details available in [16].So we use the I2[17] to engage the base code in the process through the open to clause. Providing a means for the base code to be involved in regulating advising crossing its boundary using the image construct enhances over all system robustness. The image as it is allows advising without aliasing on all methods for simplicity since this example focuses more on advising constraints.

VI. CONCLUSIONS

This work attempts to review the AOP shortcoming and provide a solution to make more room for AOP adoption. The interface Image (I2) approach [17], an attempt at a design geared toward solving the AOP modularity problem. We provide a language-level solution to both problems in the form of a new construct added to classes. The construct exports a view of the advisable class interface for aspects to refer to instead of member method signatures directly. Additionaly, the new image construct allows classes to attach advising constraints to joinpoints guiding advice applications coming from the aspect side. Another interesting extension, is allowing interfaces and aspects to declare images. Allowing aspects to declare images should help to reduce coupling among aspects that advise each other. We argue that interface images development can benefit from tool support. We believe that the ease of use of interface images will bring more adoption of AOP into the software engineering community.

VII. ACKNOWLEDMENTS

We place on records and wish to thank the author Mohamed I Elbendary PhD. Software Design Engineer, FasTraK Softworks, Inc., for providing insight about AOP shortcomings and useful research on Aspect-Oriented Software Systems Enhancing Modularity.

REFERENCES

[1] Ivan Jacobson. Aspects: The missing link. Software development Magazine, November 2003.
[2] Ivan Jacobson. A case for aspects. Software development Magazine, October 2003.
[3] Gary T. Leavens and Curtis Clifton. Multiple concerns in aspect-oriented language design: A language engineering approach to balancing benefits, with examples. Technical Report TR 07-01a, Iowa State University, 2007.
[4] J. Kienzle and R. Guerraoui. Aop: Does it make sense? the case of concurrency and failures. In 16th. European Conference on Object-Oriented Programming (ECOOP'02), pages 37–61, 2002.
[5] John Boyland. Remote attribute grammars. Journal of The ACM, 52(4):627–687, July 2005.

[6] Curtis Clifton, Gary T. Leavens, and James Noble. Mao: Ownership and effects for more effective reasoning about aspects. In European Conference on Object-Oriented Programming 2007 (ECOOP'07), pages 451–475, 2007.
[7] Adrian Colyer. Aspect-Oriented Software Development, chapter AspectJ, pages 123–143. Addison-Wesley, 2005.
[8] The AspectBench Compiler. The aspectbench compiler for aspectj.http://www.aspectbench.org/.
[9] Siobhan Clarke and Robert J. Walker. Aspect-Oriented Software Development, chapter Generic Aspect-Oriented Design with Theme/UML, pages 425–458. Addison-Wesley, 2005.
[10] R. E. Filman and D. P. Friedman. Aspect-Oriented Software Development, chapter Aspect-oriented Programming is Quantification and Obliviousness, pages 21–35. Addison-Wesley, 2005.
[11] W. G. Griswold, K. Sullivan, Y. Song, Y. Cai, M. Shonle, N. Tewari, and R. Hridesh. Modular software design with crosscutting interfaces. IEEE Software, pages 51–60, 2006.
[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G.Griswold. An overview of aspectj. In 15th. European Conference on Object-Oriented Programming (ECOOP'01), pages 327–354, 2001.
[13] I. Kiselev. Aspect-Oriented Programming with AspectJ. Sams Publishing, 2003. ISBN 0-672-32410-5.
[14] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In Proceedings of the 27th. International Conference on Software Engineering ICSE'05, pages 49–58, 2005.
[15] G. Kniesel and T. Rho. Generic aspect languages - needs, options and challenges. In Proceedings of La deuxime Journe Francophone sur la Programmation Par Aspects JFDLPA 2005, pages 1–20, 2005.
[16] Ramnivas Laddad. AspectJ IN ACTION, Practical Aspect-Oriented Programming. Manning Publications Co., 2003. ISBN 1-930110-93-6.
[17] Mohamed I. ElBendary, "Enhancing Modularity in Aspect-Oriented Software Systems", Ph.D Thesis, The University of Wisconsin–Milwaukee December 2008

AUTHORS PROFILE

Kotrappa Sirbi having M Tech (CSE) from Visvesvaraya Technological University Belgaum, M S (Software System) from BITS, Pilani and B .E (EE) working with Departmenent of Computer Science & Engineering, K L E's College of Engineering & Technology, Belgaum since 1985.Presently working with K L E's B C A, Belgaum (Deputation).He has 06 International Journal publications,02 International conference papers and 02 National Conference papers and 02 workshop papers for his credit. His areas of interest are Software Engineering, Object Technology and its evolution like. Design Patterns, AOP (Aspect Oriented Programming).He is member of ISTE, Members of CSTA, ACM, USA and Member of IAENG.

Prakash Jayanth Kulkarni having Ph.D (Electronics) and M E (Electronics) by Research from Shivaji University, Kolhapur and B.E (Electronics & Tele) from Poona University, Poona working with Walchand College of Engineering, Sangli since 1981 and 1980-81 worked with Trans Lines Division,M S E B haing 13 international conference papers and 10 National conference papers and 10 journal papers. His areas of interest are Digital Communication, Digital Image Processing and Computer vision, Software Engineering, Artificial Neural Network and Genetic Algorithms. In 2001 he received a distinguish Samaj Shree Award for rendering service to society.