A Family of Aspect Dynamic Weavers

Wasif Gilani
Olaf Spinczyk
University of Erlangen-Nuremberg
Martensstr. 1
D-91058 Erlangen
Germany
{wasif, spinczyk}@informatik.uni-erlangen.de

Abstract

Aspect-oriented programming is today mainly promoting the approach of applying aspects statically by means of preprocessors. We do find some work towards applying aspects dynamically but only limited to specific environments and no work so far has been directed toward our domain of interest which is deeply embedded systems characterized by very small memory and power. Also the work undertaken so far does not take into consideration the family-based approach since we have drastically different environments according to resources and one should be able to build up his own dynamic weaver from the family of weavers best suited for his environment. The paper presents an approach to achieve dynamism in applying aspects and also towards developing family-based aspect weavers to this problem which would be best suited for any sort of environment rather than having specific environment restrictions.

1 Introduction

To get maximum reuse of code and design, it is necessary to achieve complete separation of cross cutting concerns from the main functionality of an application. Following the aspect-oriented approach, the main functionality of any application is normally captured in classes and the cross cutting concerns are captured in separate aspects. Thus, in general case, we can say that the software systems developed using aspect-oriented programming techniques consist of classes and aspects whereas classes mainly implement the main functionality of an application, for example, put and get functionality in the case of bounded buffer problem, managing stocks or calculating insurance rates. Aspects, on the other hand, capture cross cutting concerns like synchronization, tracing, persistence, failure handling, or communication. Aspects are basically used to implement global policies in a system.

In the beginning, most of the work was based on static weaving which means adding aspects specific statements at join points to the classes statically at compile time. Static weaving produces well-formed and highly optimized woven code whose execution speed is comparable to the code written without AOP. There are certain environments where it is needed to be able to change the global policies implemented through aspects during run-time. Thus static weaving is not sufficient for such applications because in static weaving approach it is difficult to later identify aspect-specific statements in the woven code. Thus, it will be time consuming to adapt or replace the aspects dynamically during runtime or sometimes not possible at all. Although this flexibility is not a requirement in all scenarios, there are scenarios where applications could benefit from it.

Dynamic weaving means that aspects can be applied or removed at any time during runtime. Thus dynamic weaving allows the integration between components and the aspects at run time,

resulting in a system which is more adaptable and extensible. In short, in dynamic weaving aspects can be added to the system on the fly and, thus, can help avoid re-compiling, re-deployment and re-start of an application [5]. There are cases where it is more useful and flexible to have dynamic weaving as compared to the static weaving. One scenario would be where a load balancing aspect [4] could replace the load distribution strategy woven before with a better one depending on the current load of managed servers. Therefore in certain cases survival of aspects at run time is necessary to allow them to adapt to suitable policies according to execution time information. Another example could be addition of debugging aspect to get some particular data from some long running embedded application. It would be more flexible and beneficial to just add this debugging aspect to the application while running and get the required information than to stop the application and restart it again. Another case could be of the tracing aspect which could be deployed dynamically in some software system where some malfunctioning has happened. Another case where dynamic weaving provides edge over static weaving is hot fixes in web applications [10]. A hot fix is an extension applied to a running application server to modify the behaviour of a large number of running components. Another interesting example is the adaptation of mobile devices [10] since mobile devices have very small memory and so the device is not able to have all the software components needed in various locations from the start and so it should dynamically acquire the needed functionality it needs in a certain location and discard when location changes. This functionality has often a cross-cutting character and so is realized as aspects.

This paper starts with analyzing different existing dynamic weaver infrastructures to systemically describe the variabilities of this domain. The section 3 of this paper discusses the idea of program family concept and applying this idea towards family-based development of dynamic aspect weavers. In section 4 design methodology of our approach will be discussed. Section 5 describes one specific case of constructing dynamic weaver from the selection of certain features from feature model. Section 6 concludes the paper.

2 Dynamic Weaver Infrastructure

A run-time system is needed to support the weaving of aspects dynamically. There is not much research work going on in the direction toward building dynamic aspect weavers in the C++ domain. We do find some work but mainly in Java which is not suitable for our domain of interest which is embedded systems characterized by very small memories. Also focus has been on the development of single application specific weavers rather than families of weavers so there is no chance to make an optimized use of these weavers for any particular environment. We propose to build a family of dynamic aspect weavers to cover a whole range of environments.

Existing dynamic aspect infrastructures can be differentiated according to the way the functionality class is bound to the dynamic aspect weaver and the varying support provided by these for dynamic weaving. So far there have been three main approaches namely proxy-based, interpreter based and binary code manipulation.

In the proxy based approach [6], a proxy is used to intercept the incoming requests to the functionality class. The role of so called weaving is performed by an object called AspectModerator. All the aspects are registered with this AspectModerator object and proxy uses AspectModerator object to evaluate the aspects for every method of the functionality class. When the request arrives in the system, it is intercepted by the proxy and if the request is for the creation of an aspect then the proxy first checks whether this aspect is already registered with the AspectModerator object. If not, the proxy calls AspectFactory to create an aspect. The proxy then registers newly created aspects with the AspectModerator object. In case of a request for method

invocation, the proxy calls the AspectModerator object to evaluate the aspects associated with this invocation. In this framework approach focus is only on reuse. This is not applicable for embedded systems because of excessive use of the design patterns with expensive abstract classes and virtual functions.

In interpreter extension approach, standard interpreter is transformed to build a new interpreter (plug-ins). This approach is normally applied by using Java virtual machine (JVM) since in C++ domain no interpreter is used. The interpreter is extended to check for the events happening in the execution and on happening of the events; advice code is applied as methods. Prose [5] is based on locating support for weaving and unweaving of aspects directly in the JVM. It is a JVM extension which can intercept calls at run-time. It makes use of the Java virtual machine debugging interface (JVMDI). It provides a new API within the JVM for weaving aspects at run-time called Java virtual machine aspect interface (JVMAI) which is designed as a plug-in to JVM. Thus applications must run with Prose specific JVM. Axon [12] is also based on interpreter extension approach and makes use of the debugging interface of JVM. It is an ECA (Event-Condition-Action) rules inspired model. Pointcuts are described in terms of events (join points) and conditions, while advice is described as an action associated with events and conditions. Axon is also implemented as a plug-in to Sun's JVM. Events are generated by the JVM and the conditions are checked by Axon via run-time inspection with the JVM debugging interface and the actions are advices which are implemented in plain Java methods. In this model around advice (AspectJ [2], AspectC++ [1]) is not feasible because it is based on interception at join points and there is no way an advice could replace the called method. Axon supports limited join points namely method entry/exit, exception throw/catch and field read/write. Both Prose and Axon are slow as the debugger imposes a certain overhead in the execution of applications. The JVMAI approach in Prose does not support crosscuts that add new members to a given class in the original code (introductions), because its implementation cannot change the source-code or byte-code of the original application. The debugger in current implementation of Prose is no longer available for use for debugging applications. Some of the approaches based on JVMDI are being modified to make use of just-in-time compiler instead of debugging architecture to improve the performance. Prose2 [10] is implemented by extending the VM's just-in-time compiler and no longer uses debugger architecture.

The binary code manipulation approach has mainly been employed in Java (JAC [7], Wool [11]). Most of these available approaches make use of JVMDI or changing the byte code at load time. Hooks are either inserted statically in all join points or inserted into the program at run-time "just-in-time" when the programmer directs the program to start using an aspect. Approaches where JVMDI is employed, the debugger interface allows a user to stop and query the state of a running program. These techniques are slow as debugger imposes certain overhead (context switches) on the execution of applications. Moreover these different approaches which make use of the JVM vary in terms of the join point support and performance. Wool [11] supports only method calls, field accesses, object instantiation, and exception handlers and does not allow introductions. JAC changes the byte code of classes of an application when they are loaded into the JVM and so have a high number of empty hooks and only support method calls and exception throws. In the C++ programming domain we are not aware of any previous work in aspect languages which provides dynamic weaving except for the microDyner [3] approach which was developed originally for the C language. This approach is processor specific (Pentium architecture) and does not allow more than one aspect to affect the same join point. In the microDyner approach, which makes use of binary code manipulation, aspects are deployed dynamically at run time in C programs. It realizes the weaving process by directly rewriting the code being executed. The microDyner approach is being extended to support C++ [8]. However, still this approach supports one aspect per join

point and also aspects are not objects in this approach and so we cannot make use of nice features of object-oriented programming like constructors, inheritance etc.

These different dynamic weaving approaches differ from each other in a number of ways. Some of these approaches offer a limited set of join-points, thereby limiting the amount of application features an aspect can adapt. Some support just code join points while others support introductions as well. Also these different approaches differ in terms of support for either single or multiple aspects per join point and advance knowledge of aspects etc.

In this section different existing dynamic weaving approaches have been analyzed to find out their commonalities and variabilities. All of these different approaches concentrate on satisfying the requirements for particular systems and offer different performance penalties like execution speed, memory consumption and join point support etc. For example, proxy-based weaver is not suitable for domains where there is a very small amount of memory and run-time resources because of extensive use of abstract classes and virtual functions. Also for such domains like embedded systems, which are very short of memory and run-time, it is not possible to use JVM based weavers because of the memory space occupied by the JVM. Program family concept implements the idea of building application specific systems and uses feature domain analysis to represent the commonalities and differences between the applications in a whole domain. In the next section, program family concept will be described along with its application in the field of aspect dynamic weavers.

3 Applying Program Family Concept

The main aim of our work has been to shift focus from the development of single dynamic weavers to the families of weavers. A set of programs is considered to be a program family if they have so much in common that it pays to study their common aspects before looking at the aspects that differentiate them [14]. Domain engineering [9] helps us to accomplish this goal. Domain engineering moves the focus from the code reuse to reuse of the analysis and design models. Domain analysis is the first phase of domain engineering which involves the process of systematic organization of the existing domain knowledge in a way that enables and encourages the extensions to be made in creative ways. The next phase of domain engineering is domain design which involves the development of an architecture for the family of systems in the domain and to devise a production plan. The last phase of domain engineering is domain implementation which involves implementing the architecture, the components, and the production plan using appropriate technologies. The results of the analysis, collectively referred to as a domain model, are captured for reuse in future development of similar systems. A domain model is an explicit representation of the common and the variable properties of the systems in a domain. Feature models define a set of reusable and configurable requirements for specifying the systems in a domain. A feature model consists of a feature diagram and some additional information. Features and feature models [9] are used to capture the commonalities and variabilities of systems in a domain during domain analysis. A key part of feature model is a feature diagram. A feature diagram represents a hierarchal decomposition of features including the indication of whether or not a feature is mandatory (each system in a domain must have certain features), alternative (a system can possess only one feature at a time) or optional (a system may or may not have certain features).

Some applications may require only a subset of services or features that other applications need. These ‘less demanding’ applications should not be forced to pay for the resources consumed by

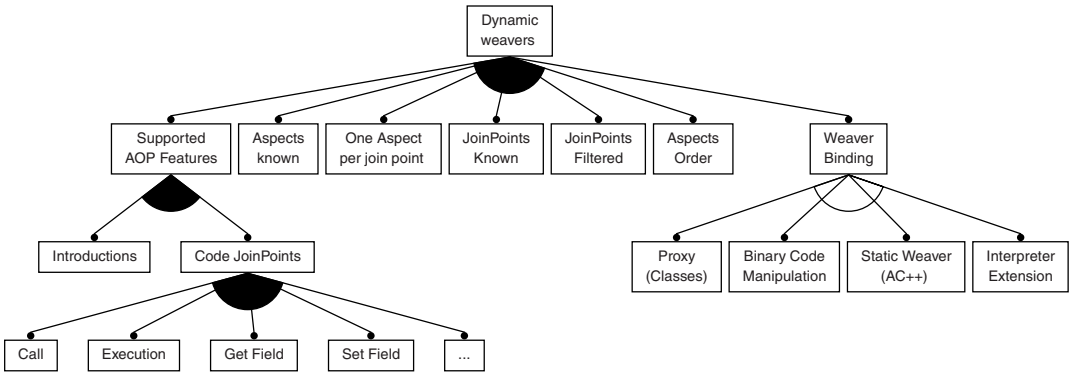

Figure 1. Feature model for Dynamic Aspect Weavers

unneeded features [14]. In a program family concept a minimal subset of system functions provides a common platform of fundamental abstractions. These minimal subset of functions capture common functions that are useful to build specialized systems. The mechanisms from which more enhanced system functions can be derived are called minimal basis. A stepwise functional enrichment of the minimal basis is performed by means of the minimal system extensions. These extensions are made on the basis of an incremental system design, with each new level being a new minimal basis for additional higher level system extensions. Since the extensions are made only on demand thus a true application oriented system evolves.

Different applications can have different requirements from the dynamic weavers. We hold the view that less demanding applications should not be forced to pay for the resources consumed by the unneeded features. It is hard to imagine if any of the existing dynamic weavers could be used in embedded systems where applications are executed under extremely limited resource constraints. Thus weavers are required to be designed to specifically support the execution of applications under such resource constraint environments. In consideration of the specific demands of these applications, it becomes extremely difficult, if not impossible, to successfully adapt existing weavers. The program-family concept is being applied to create application specific dynamic weavers. A family-based dynamic weaver targets a wide range of applications including embedded systems. The program family concept does not dictate any particular implementation technique. In our proposal, based on program family concept, the dynamic weaver is incrementally enriched by minimal extensions. The weaver extensions are customized with respect to specific user demand. Thus applications are not forced to pay for the resources that will never be used. One of the main reasons for applying family-based design is to achieve the desired application orientation and to reduce the memory and run time consumption. The goal is to support applications with their desired specialized family member which provides all necessary functionalities but omits any features which are not required.

In the rest of this section domain analysis is performed for dynamic aspect weavers and a feature model is drawn to capture the commonalities and variabilities of dynamic weavers.

3.1 Dynamic Weaver Construction Set

We have made an effort to develop a feature model for dynamic aspect weavers consisting of a number of features. The output from this feature model is a family member which would be a dynamic aspect weaver. Use of feature model makes it easy to generate the family members and to handle the complexity of configuration in a better way. A feature model for "Dynamic aspect weavers" is shown in figure 1. It is a quite simple model and allows specifying the required binding mode of the aspect weaver from the four available binding modes. Whether a dynamic weaver's binding with functionality class is proxy-based, binary code manipulation, interpreter extension or static weaver based is up to the specific application requirements. In any dynamic aspect weaver it is assumed that only one of these binding possibilities would be used as shown in the feature model. The alternative features for dynamic weaver are indicated in figure by an arc which is not filled. Then after there can be lot of different scenarios regarding requirements for specific applications about before hand knowledge of aspects, aspects not known in advance, join points, order of activation of aspects, join point filtration, supported join points etc. Keeping in view of different possible requirements different dynamic weavers can be constructed by the selection of different features. In the next sections all these different possibilities will be discussed along with the possible feature selection scenarios.

3.1.1 Aspects Known

There can be variability in the dynamic weaver construction regarding advanced information of aspects going to affect the join points. This feature "Aspects Known" from the feature model is selected in case when, in certain applications, it is possible to have an advanced knowledge of the aspects. In general case when this feature is not selected then there will be checks for all join points to find out if there are aspects registered for them irrespective of the possibility that for some join points there might not be any aspects registered at all. So in case even no aspect is registered, all checks, whether or not the system should execute advice, are performed. This situation will result in unnecessary method calls which involve large overhead.

It is possible to get rid of these types of checks for join points for which there are no aspects registered by selecting the feature "Aspects Known" in the weaver construction. This feature selection means that there is advance knowledge of aspects. When aspects are known in advance it would be much more efficient to only register the join points which are going to be affected by these aspects than to register all the join points and in result resources can be saved. Join points which are not going to be affected can be inlined. Now in this dynamic weaver construction the runtime infrastructure will be consumed for limited join points which are registered with the run-time system (only for which there are aspects registered) and so the system will be much more efficient.

If there are more aspects which are going to affect the same join point then normally lists are maintained against each join point containing ***before*** and ***after*** advices. So when there is an advance knowledge about the number of aspects going to affect each join point then it would also be possible to fix the size of advice lists associated with each join point and hence saving space. Another important benefit from the advance knowledge of aspects is that their order of execution can be resolved statically and so run-time infrastructure can be saved. Example where this can be implemented is having some system and there are three policies of security to be implemented by means of aspects. Also if we have an application in which more than one aspect can affect the same join point then it means it is essential to select the "Aspect Order" feature as well, while

dynamic weaver construction, to resolve the conflicts between different aspects affecting the same join points.

3.1.2 Aspects Order (Interaction)

There are some dynamic weavers which are restricted to support only one aspect per join point [8]. In other cases we have frameworks [6] which support any number of aspects affecting the same join point. If more than one aspect (advice) affects the same join point and there is dependency between the advice codes then it might be necessary to define an order of advice execution ("aspect interaction") in order to avoid conflicts. The order of activation is supported in static weaving technologies like AspectC++, AspectJ. In Netinant's aspect-oriented framework, the order of activation of aspects is predefined, it is defined that the synchronization aspect has to be verified before the scheduling aspect. If security aspect is introduced then it is needed to be handled before the synchronization aspect. A possible reverse in the order of activation of the aspects can violate the semantics. Moreover it is possible to alter the order of activation on the fly.

In certain dynamic weavers, like microDyner, it is not allowed that more than one aspect can affect the same join point. Thus in such dynamic weaver constructions there is no need to select the feature "Aspects Order" which is only required when dynamic weaver is supposed to support multiple aspects per join point. Also in such cases there is no need to maintain lists of before and after advices against each join point registered with the run-time system and so there would be no need to traverse the whole lists during runtime to invoke advices dynamically and as a result weaver will be much more efficient and fast.

3.1.3 Join Points Known

In the case of join points there can be again two possibilities, first being that all affected join points are known in advance and second that there is no advance knowledge of the join points going to be affected by the aspects. In the case of join points known in advance, once the system starts running it will be affected by the aspects which are already into the system. If the system is extended (classes are loaded incrementally) then the aspects are not able to affect these additional loaded classes (aspects do not have to affect the code loaded later in the system). In other words, additional classes are allowed to be loaded if it is known that no aspect would be affecting join points in these classes. When join points to be affected are known in advance, compile time matching can be done resulting in saving of run time resources. In example we can think of a system in which we know join points which are going to be affected in advance and so we are able to apply different versions of security policies, implemented as aspects, to this system. Now if the system is extended, aspects would not be affecting the additionally loaded classes.

3.1.4 Join Points Filtered

Join points can be filtered, for example, by means of pointcuts according to the varying requirements. In some systems there might be requirements to apply aspects in specific modules. For example we can have two systems, in one system aspects can affect the whole system but in the other case we can do filtration and so only specific module will be affected by the aspects. There is no need to change the aspects, only the behaviour of the aspect is modified but the implementation remains same.

3.1.5 Supported AOP Features

To which join point in a system is it possible to apply aspects? There are different approaches and these vary either according to the type of join points supported by them or if they support introductions. Code join points can be defined as method calls, method executions, set field, get field etc. Introduction is how modification is done to a program's static structure, namely, the members of its classes and the relationship between classes. Introductions are used to extend program code and data structures in particular. Most of the join points are supported by AspectJ and AspectC++ like method call, method execution, constructor call, constructor execution, object initialization etc. Both AspectJ and AspectC++ also support introductions. There are other approaches which support specified set of join points like Wool [13] supports only method calls, field accesses, object instantiation, and exception handlers. Axon [12] supports method entry/exit, exception throw/catch and field read/write join points. JAC' supports method calls and exception throws [7]. Wool does not allow introduction since the HotSwap does not allow reloading a class file to which a new method or field is appended. Similarly Prose [5] does not support introductions. There are certain applications where it might be required to have support for introductions and thus dynamic weavers would need this feature selection. One example where introduction could be needed to have support is when pointers could be added to objects statically and then during run-time data could be added to these objects.

This is still not a very comprehensive feature model. We are still working on it and there are many more common and variable features of dynamic weavers which are needed to be represented in this feature model.

4 Design Methodology

Our proposal is based on promoting two ideas. First a dynamic weaver should be able to make use of both the static and dynamic weaving according to the specific application requirements and cost considerations. Secondly the design should be based on the family-based approach. The reason for having support of static weaving in the dynamic weaver construction is logical if we consider the advantages we get from static weaving in terms of performance of static weaving as compared to dynamic weaving. Ideally, an implementation should support both. Aspects that don't need to be adapted at runtime should be woven statically for performance reasons since our dynamic aspects do consume run time resources and so dynamism should be allowed for aspects which do have runtime changing behaviour or which need to change policies during runtime. Even in the case of byte code manipulation, users can be allowed to choose a suitable hook at each join point considering the whole cost. Either they can be inserted as a breakpoint by a debugger and so executed by debugger or can be embedded as a method call using dynamic code translation [12]. For example an application where we could think of having support for both static and dynamic weaving together is when an aspect such as authentication [13] can be woven statically as it is very unlikely that an authentication policy changes during program execution. On the other hand, a scheduling policy has to be adapted most likely at run-time. Scheduling can therefore be viewed as a dynamic aspect. A good mix of the dynamic weaving and static weaving promises to improve AOP effectively. Aspects can be efficiently tested by dynamically inserting them, checking the behavior of the application and then removing the aspects to perform corrections. In case of adding an aspect statically it is required that the running application be

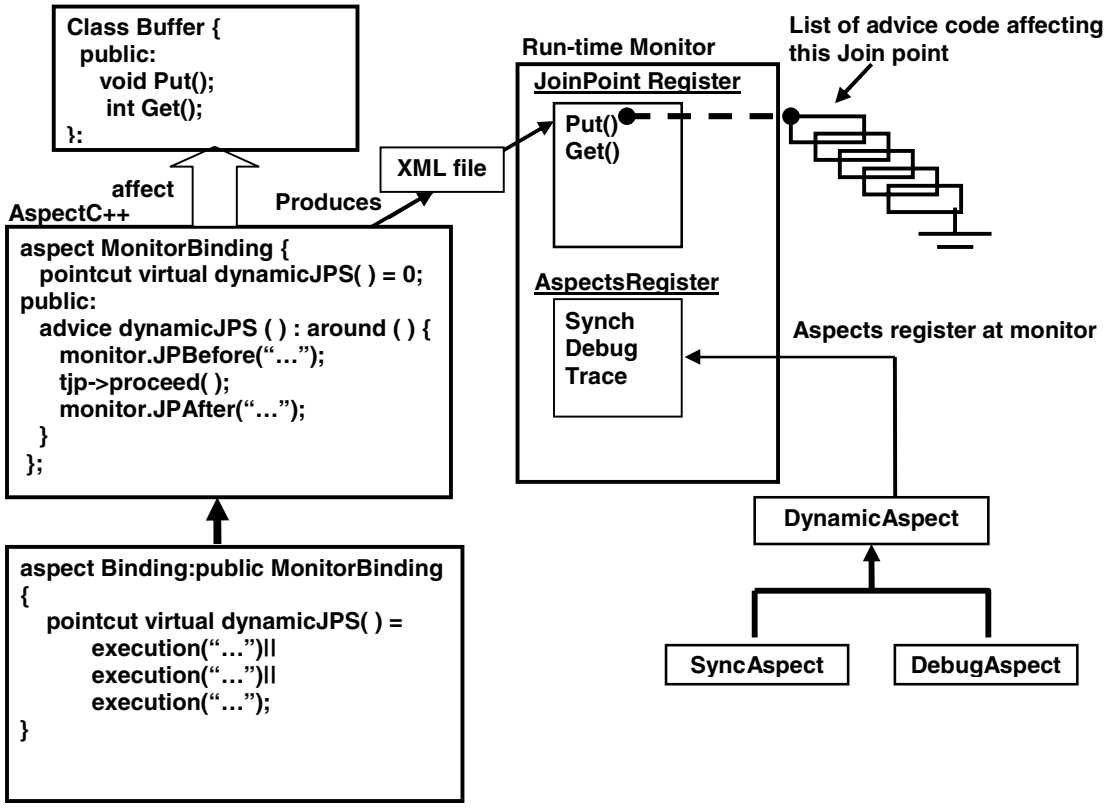

Figure 2. Aspect Dynamic Weaver Architecture

shut down thereby losing all the run time data. Eventually once the aspect code is stable, the aspect can be woven through the application code using a static weaver to improve performance [5].

5 Weaver Instantiation from Family of Dynamic Weavers

In this section an instantiation (construction) of a dynamic weaver will be described from the family of dynamic weavers (feature model). This weaver would be constructed in way that it would be able to support static as well as dynamic aspects and also it would be able to support multiple aspects per join point. As can be seen from the feature model, there is a limit to select only one binding mode feature of the dynamic weaver to the functionality class. This particular construction makes use of selection of feature "static weaver (AspectC++)" as a binding mode to the dynamic weaver (Figure 2). Though AspectJ is also an option which is a complete and powerful language extension for aspect-oriented programming but the costs (run- time and code size) of Java run-time-environment are not feasible for deeply embedded systems which are having very low memory constraints. The selection of binding feature is totally independent of the functionality class. Dynamic aspect weavers can be constructed by selecting any of the binding modes available. In our construction a dynamic weaver is constructed from three main modules which are independent of each other and have clearly defined interfaces. These modules are:

- Run-time monitor
- Aspect binding (Dynamic Aspects)

- Weaver binding (Static Weaver)

The run-time monitor plays a central role in this dynamic weaver. The main role of the run-time monitor is to register the join points and the aspects. It also co-ordinates interaction between the aspects and the functionality class. In this dynamic weaver construction, it is assumed that more than one aspect is able to affect the same join point and so the feature "Aspects Order" is selected. The source file for run-time monitor for registering join points is an XML file which is generated by the static weaver (AspectC++). This file could be used statically to register join points in the case join points are known in advance. In embedded systems one rule is followed which is to do as much processing as possible before run time, creating a run-time environment that is as efficient as possible. In case of join points not known in advance we can convert this XML file to a binary format to make the processing efficient and thus improving the system's performance. The run-time monitor also has a task to take care of order of execution of aspects in the case if there are more than one aspect interested in the same join point.

There are two types of aspects which are supposed to be supported by this dynamic weaver. First being the static aspects defined in a specialized aspect description language like AspectC++ or AspectJ. Secondly, the aspects which are dynamically invoked during run time and are C++ classes. Each class is supposed to have two advices which are simple methods. One is "**JPBefore(…/*method id */…)**" and other is "**JPAfter(…(…/*method id */…)**" and both of these advice methods take method identity as a parameter and it is the run-time monitor which invokes these methods in static weaver.

The static weaver (AspectC++) has been developed by the authors and it is a general purpose aspect-oriented extension of C++, modeled following the approach of AspectJ. AspectC++ is implemented as a C++ preprocessor based on PUMA [1]. PUMA is a source code transformation system for C++. The output of this preprocessor is the C++ source code with the aspect code woven in. Afterwards a conventional C++ compiler is used to get the code translated to the executable code.

In this aspect dynamic weaver, the functionality class is bound to the framework with a static weaver (AspectC++). The static weaver produces XML file as a result of the interaction with the functionality class. This XML file consists of all the information of the join points contained in the functionality class. These join points are then registered with the run-time monitor using this generated XML file. This dynamic weaver is being constructed on the idea that one should be able to decide which aspects need to be woven dynamically at run-time and which aspects to be woven statically at compile time depending on the performance. Thus there are certain aspects which don't need to be woven dynamically and so static weaver is used to weave these aspects statically to save run-time infrastructure. The dynamic aspects are normally simple classes and they are registered with the run-time monitor. As soon as some dynamic aspect is registered with the run-time monitor, the list of join points registered with the run-time monitor is traversed to find out which join points are affected by this aspect. Since in this dynamic weaver one join point is supposed to be affected by more than one aspect (here again other scenarios are possible by the selection of, for example, "One Aspect per Join point" feature), a list of advices, which are methods defined in each aspect, which are to affect a certain join point, is maintained to be executed once this join point is invoked by the run-time monitor in the static weaver.

In this dynamic weaver, run-time monitor is able to add or remove aspects on the fly. Functional classes do not know about the aspects in advance but of course it purely depends on the features selected from the feature model (figure 1). When some major dynamic aspects of the system are

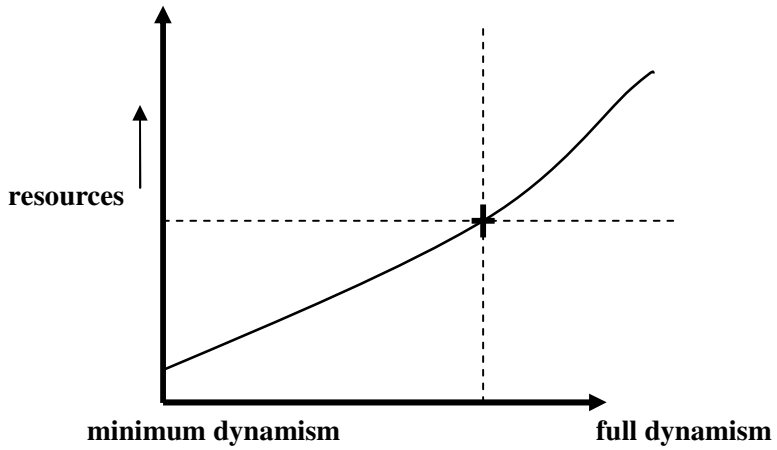

Figure 3. Resource consumption vs. provided dynamism

to be defined like scheduling, synchronization, tracing, fault tolerance, these aspects are defined as being derived from the same super-aspect. Thus this dynamic weaver promotes reusability. An abstract super-aspect provides transparency since sub-aspects can use the super-aspect without knowing the internal implementation details of super-aspect. Even new aspects can be introduced into the system without any problem. If some super-aspect is changed to add some new features, the sub-aspects don't need to be changed as far as the interface remains constant. All dynamic aspects are registered with run-time monitor with some call like:

monitor.registerAspect(Aspect SyncAspect) /*SyncAspect captures synchronization concerns

monitor.JPBefore(putId);	/* all before advices are executed when run-time /*monitor is invoked from static weaver
tjp->proceed();	/*actual method is invoked
monitor.JPAfter(putId);	/* all after advices are executed when run-time /*monitor is invoked from static weaver

In static weaver (AspectC++) there is also a possibility to derive an aspect from super aspect. Abstract aspects can be defined from which new aspects can then be redefined through inheritance, thus providing programmers with an aspect hierarchy. Using advice "around" feature of static weaver allows to first invoke before advice codes (**monitor.JPBefore(…/*method id*/…)**) associated with any join point, then invoking the join point itself (**tjp->proceed**) and then finally invoking the after advices (**monitor.JPAfter(…(…/*method id*/…)**) of the join point.

6 Conclusion

This work provides a base for developing application specific dynamic weavers. Instead of inventing a new dynamic weaver architecture, this approach provides the user with the ability to construct many of those architectures. The concept of program family has been applied to build a family of aspect dynamic weavers. A feature model has been built which provides an abstract, concise and explicit representation of the commonality and variability present in the domain of

dynamic weavers. Using this approach it is possible to build dynamic weavers with as much functionality as one application can afford. This work promotes the idea that one should not be asked to suffer for the services he does not require. The program family concept helps to create featherweight weaver abstractions. These abstractions can be used by the user to construct a number of dynamic weavers.

The main goal of this approach is to be able to construct application-specific dynamic weavers by selecting only those features from the feature model which are required. The example of a specific dynamic weaver instantiation from a family of dynamic weavers has clearly demonstrated that it is possible and feasible to construct a weaver according to the specific requirements of a particular application. Figure 3 illustrates that as we move towards more dynamism in building the dynamic weavers, we have to pay more in terms of resources. The goal, while constructing a dynamic weaver for any application, is to get to a point as shown in figure where we are not exhausted of the resources and we have as much features added to our weaver construction, to support dynamism, as possible. A simple example where we could think of constructing customized dynamic weaver would be of some embedded system with very small memory in the range of, for example, 30 Kbytes. Now while doing application specific construction of a dynamic weaver for such systems, where the join points are normally known in advance, we can select features from the feature model to have as much degree of dynamism as memory space allows. The result would be a dynamic weaver which would be able to fully utilise the available memory space and allow us with as much dynamism as we can afford.

References

[1] Spinczyk Olaf, Gal Andreas, Preikschat Wolfgang Schroeder, *AspectC++: Language Proposal and Prototype Implementation*. In OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa, USA, Oct. 2001.

[2] Xerox Corporation, AspectJ programming guide, Online documentation, http://www.aspectj.org, 2003.

[3] Marc Ségura-Devillechaise and Jean-Marc Menaud and Gilles Muller and Julia Lawall, *Cache Prefetching as an aspect: Towards a Dynamic-Weaving Based Solution*, in Proceedings of the 2nd international conference on Aspect-oriented software development, p. 110-119, ACM Press, Boston, Massachusetts, USA, Mar 2003 Web

[4] Frank Matthijs, Wouter Joosen, Bart Vanhaute, Bert Robben, and Pieere Verbaeten., *Aspects Should not Die*, Position paper at the ECOOP '97 workshop on Aspect-Oriented Programming.

[5] A. Popovici, T. Gross, and G. Alonso., *Dynamic Weaving for Aspect Oriented Programming*. In 1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands, Apr. 2002.

[6] C. A. Constantinides, T. Elrad, M. E. Fayad, Netinant P., *Designing an aspect-oriented framework in object-oriented environment*, ACM Computing surveys, March 2000.

[7] Pawlak, R., Seinturier, L., Duchien, L., Florin, G., *JAC: A flexible framework for AOP in Java*. In Reflection 2001

[8] Yan Chen, Masters thesis, *Aspect-Oriented Programming (AOP): Dynamic Weaving for C++*, August 2003, Vrije Universiteit Brussel and École des Mines de Nantes

[9] Krzysztof Czarnecki, Ulrich W. Eisenecker, *Generative Programming Methods, Tools, and Applications*, Chapter 4, Addison Wesley 2000

[10] A. Popovici, T. Gross, and G. Alonso., *Just-In-Time Aspects: Efficient Dynamic Weaving for Java.,* AOSD 2003*,* Proceedings of the 2nd international conference on Aspect-oriented software development**,** Boston, Massachusetts

[11] Yoshiki Sato, Shigeru Chiba, Michiaki Tatsubori, *A Selective, Just-In-Time Aspect Weaver*, Proceedings of the second international conference on Generative programming and component engineering**,** Erfurt, Germany, Year of Publication: 2003

[12] S. Aussmann, M. Haupt, *Axon – Dynamic AOP through Runtime Inspection and Monitoring*, First workshop on advancing the state-of-the-art in Run-time inspection (ASARTI), 2003.

[13] Shigeru Chiba, Yoshiki Sato, and Michiaki Tatsubori, *Using HotSwap for Implementing Dynamic AOP Systems*, ECOOP'03 Workshop on Advancing the State of the Art in Runtime Inspection (ASARTI), July 21st, 2003.

[14] D.L. Parnas., *Designing Software for Ease of Extension and Contraction*, IEEE Transactions on Software Engineering, SE-592):128-138, 1979.