Dynamic Aspect Weaver Family for Family-based Adaptable Systems

Wasif Gilani and Olaf Spinczyk

Friedrich-Alexander University Erlangen-Nuremberg Germany
{wasif, spinczyk}@informatik.uni-erlangen.de

Abstract. Complex software systems, like operating systems and middleware, have to cope with a broad range of requirements as well as strict resource constraints. Family-based software development is a promising approach to develop application-specific systems from reusable components. However, once statically configured, these systems still need to adapt at runtime according to the dynamics of the environment. The majority of the concerns in the complex systems, that need to be adaptable, are crosscutting. With the application of Aspect-oriented Programming (AOP), these concerns can be cleanly encapsulated, and then dynamic AOP can be applied for the adaptations to be contained, and applied at runtime. An efficient dynamic aspect weaver is needed for the dynamic weaving and unweaving of these crosscutting concerns into the system. None of the currently available dynamic weaver can be optimized according to specific application requirements. In this paper we present the family-based dynamic weaver framework that supports the static as well as dynamic weaving and unweaving of the aspects to the components. By applying the program family concept, the system itself as well as the dynamic weaver, built on top of it, is tailored down to provide only the features or services required by any particular application.

1 Introduction

The complex software systems, like operating systems and middleware, are notorious for their problematic structure due to the high degree of crosscutting concerns. These systems are, traditionally, designed and built to provide a wide feature set to suit the needs of multiple problem domains. The extra features not used by an application contribute to unnecessary code size and configuration complexity.

These conventional software systems are not suited for use in some particular environments, for example, distributed embedded environments, as they need to scale with specific requirements, pertaining both to the hardware as well as the software level. Thus, it is quite a nightmare to build a system that could fulfill all the requirements of different applications, and still will be economical in terms of resource consumption. The solution is to be able to tailor down the system so that it provides only

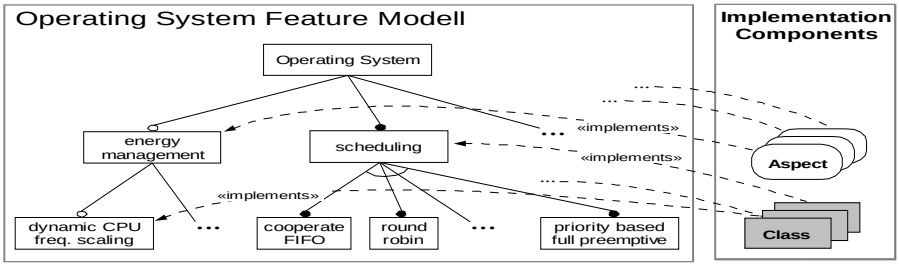


Fig. 1. Available features and their implementation artifacts

the services needed by any particular application. This leads to a product-line or family based [10] approach, where the variability and commonality among system family members is expressed by feature models [11]. A crucial point is the mapping of all selectable and configurable features to their corresponding, well encapsulated, implementation components. Many of the features that need to be adaptable or reconfigurable in these complex systems are crosscutting in nature. This means that their encapsulation is limited, as their implementations crosscut, and thus hinder adaptability and granularity in the family-based development. This makes it almost impossible to implement them as independent encapsulated entities and thereby restricts modularity, variability and granularity. Some examples of such crosscutting concerns are security, distribution, scheduling, transaction, fault tolerance, quality of service (QoS) and logging.

Aspect-Oriented Programming (AOP) [1] is applied for the localization and encapsulation of such crosscutting concerns into modules called aspects. The aspect code guides a tool, the aspect weaver, inserting code fragments specified by the aspect code into locations where they are required. These insertion points are called join points. AOP complements object-oriented programming by allowing to modify the object-oriented model, statically (static weaving) as well as dynamically (dynamic weaving), to create a system that can grow to meet the new requirements. A well directed application of AOP principles in the development of family-based systems lead to a high variability, modularity and granularity of the selectable system features, as their implementations can not only be encapsulated by classes, but also by aspects. This potentially results in very flexible systems that offer configurability of even fundamental architectural properties [2]. As an example, figure 1 shows a part of an operating system product line feature model, where features are mapped to those classes and aspects that provide their implementation.

For adding and removing the features realized as dynamic aspects in the adaptable family-based systems, there is a need of an optimized dynamic weaver. Dynamic weaver enables the dynamic aspects to be woven and unwoven from the system on the fly, which makes it useful for rapid prototyping and enables the systems to adapt their services in response to changes in the requirements. Most of the existing weavers are not suitable for all domains because of either their excessive use of resources or application-specific solutions. Here again, we apply the family-based approach and come up with a family-based dynamic weaver. An application-specific tailored

weaver is constructed from the weaver family by leaving out as much as possible and selecting only those features, which are required to fulfill the applications demands. Besides a fine-grained selection of the available AOP features (required AOP features), it is especially possible to exploit a-priori-knowledge about the system and its execution environment. This results in a much optimized, low-cost application-specific dynamic weaver.

This paper is organized as follows. Section 2 describes the family-based concept in combination with AOP. In section 3 we will be discussing some of the available dynamic weavers. Section 4 presents our family-based dynamic aspect weaver approach along with the feature model developed. In section 5, the main architecture and implementation details of our dynamic weaver are presented. Section 6 shows some results from the different variants of the weaver and their memory consumption. Section 7 concludes the paper along with some further research areas.

2 Family-based Software Development with AOP

We are shifting the focus from the development of single software systems to a family of systems (product-line). The same holds true for the dynamic weavers as well, which are built on top of such systems, to support reconfiguration of features realized as dynamic aspects. Family-based development allows supporting applications with their desired specialized family member, which provides all the necessary functionalities but omits any functionality or service not required by the application. This makes it possible to achieve the desired application-orientation, and to reduce the memory and run time consumption.

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 [10]. Domain engineering [11] helps us to accomplish the family-based software development. Feature-oriented domain analysis is performed to capture the commonalities and variabilities of the systems in a domain. A feature model 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). These different feature types are explained in figure 2. There are two stages where AOP is applied to family-based systems for adaptation.

2.1 Static Weaving for Static Adaptation

The user selects from the features presented in the feature model, and set defaults according to the needs of the application. A variant management system is used to specify default dependencies in the feature model to prevent the combinatorial explosion of the variants [13]. It provides a graphical user interface, which displays a hierarchical representation of the feature model of the product family. The user selects

feature nodes, which are mapped onto implementation components. This process of selecting features and setting defaults to generate a member of the family of products is known as application engineering [11]. This information is fed to generators, which output and build the final product. The static aspects are superimposed onto the primary functionality in an additive manner without altering the existing architecture. These aspects, which are woven, cannot be removed or reconfigured later during the runtime. The result of static adaptation is an application-specific product, which contains only those features, which are needed by the application. The whole process is completely automated and does not require any hand coding.

2.2 Dynamic Weaving for Dynamic Adaptation

Once the system starts running, it may be subject to the changing requirements during runtime. This is especially true in complex distributed systems, which exhibit strong dynamics. Several approaches have been adopted to achieve dynamic adaptation and reconfiguration of the software systems during runtime. Some try to provide adaptability by using patterns in several features [5]. However the customization resulting from this approach is still unsatisfactory as it leaves hooks in the core code, and null strategies substitute for the excluded features. This adds to the complexity of code as well as to the memory footprint. Other approaches suggest the use of reflection and component frameworks [6, 7, 8]. In some of these approaches, the system implementation adapts itself according to the changed environment by means of selecting different implementation strategies. These approaches mainly address the customizability and adaptability aspect of the systems. The drawback of these techniques is that these have rather large memory requirements and also incur performance overhead.

Dynamic weaving is a natural choice for implementing an adaptable system due to the reason that it can apply code retrospectively to a running application [9]. Dynamic weaving helps avoid the recompilation, redeployment and restart of the application. The dynamic adaptation of complex software systems is generally dependent on policies, which all tend to be crosscutting concerns, and, hence are realized as dynamic aspects. Dynamic weaving is basically an autonomous policy coordination facility that allows system to continuously adapt itself to a changing environment by determining which policy needs to be changed and how policies are recombined so that the system can keep performing well. Thus, for performing the job of weaving and unweaving of features, realized as dynamic aspects, the dynamic weaver is an integral feature in the feature model of our family-based adaptable software systems.

3 Related Work

Different approaches have been proposed by the AOSD community for dynamic weaving. Most of the existing approaches target the Java domain. These are generally based on Java-specific APIs, JVM Debugging Interface, static instrumentation, runtime byte code manipulation or virtual machine extensions [3, 12, 14, 15, 16, 20, 27].

Most of these Java-based approaches offer different performance penalties like execution speed (execution in debugging mode), memory consumption and join point support etc.

We are working in the C/C++ domain, and are more interested in the work being carried out in the C/C++ domain. There are several approaches in the C domain [17, 24, 25, 26]. In these approaches, hooks are inserted into the base program at weave time at all affected joinpoint positions. Arachne [17], TOSKANA [24] and TinyC^2 [26] are all based on binary code manipulation. These approaches use debugging information or symbol tables, produced by the compiler, to rewrite the binary code dynamically to inject the aspects. New version of Arachne is able to do the instrumentation of the binary code at weave time. TinyC^2 makes use of Dyninst instrumentation system for supporting runtime weaving. TOSKANA is designed specifically for in-kernel functions and is being employed for supporting autonomic computing functionality via dynamic aspects for operating system kernel.

We know only one approach which is specifically targeting C++ domain [18]. In this approach, aspects are woven by registering them against a runtime registration system. The original C++ code is instrumented, either by hand or with the help of tools, to call the runtime system at each potential joinpoint. The runtime system then calls all aspects registered for this joinpoint. There are also some proposals in the C++ domain for making use of low level virtual machine running on top of a microkernel system in cooperation with an aspect deployment service [25].

Even the approaches proposed for the C/C++ domain have some limitations. This is especially true for the DAO C++ approach [18], where the runtime system has to be called at each potential joinpoint and, thus, result in large performance overhead. Also DAO C++ supports only a limited joinpoint model, namely the execution of before and after advices. This system could be improved by making use of some joinpoint filtration mechanism to instrument only the joinpoints of interest. The binary code manipulation approaches have better performance since the overhead due to the static insertion of hooks is minimized. But these approaches have other shortcomings like they are machine and compiler-specific solutions, and therefore not applicable for the broad spectrum of hardware platforms in the domain of, for example, embedded systems. These approaches are further restricted to the amount of available symbolic information in the executable code. Moreover, all of these approaches provide a fixed runtime support system. This means that they cannot be scaled according to the requirements of any particular application.

4 Family-based Dynamic Weaver

Different applications can have different requirements from the dynamic weavers. In consideration of the specific demands of certain applications, it becomes extremely difficult, if not impossible, to successfully adapt any of the existing weavers. Thus, the dynamic weavers are required to be designed to specifically support the execution of applications under any sort of environmental constraints.

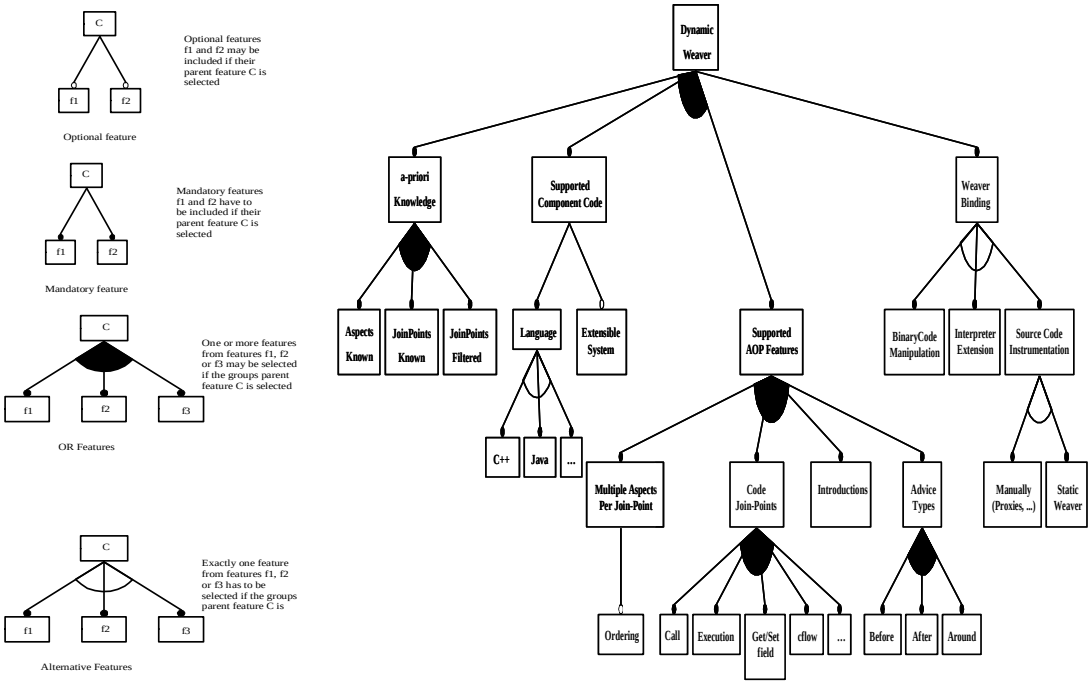


Fig. 2. Feature diagram of Dynamic Weaver

A family-based dynamic weaver is able to target a wide range of applications including embedded systems with very small memories. Applying program family concept, a domain analysis is performed for dynamic weaver's domain. A feature diagram is drawn to capture the commonalities and variabilities of the weavers as shown in figure 2. By applying the program family concept, not only the system itself, but the dynamic weaver as well, is tailored down to provide only the services or features required by a particular application, resulting in a very economical and application-specific solution. "Less demanding' applications are no more forced to pay for the resources consumed by unneeded features.

The family-based weavers are based on the technique of runtime aspect registration, but can be tailored down according to the specific application requirements. The dynamic weavers are constructed from the feature model by selecting only the required features. Each selected feature has certain cost associated with it in terms of the runtime and memory, and hence selection of features is totally dependent upon the specific application requirements and the memory available.

Our approach makes it possible to build low-cost dynamic weavers by exploiting the before hand knowledge "***a-priori-knowledge***" about the system (domain analysis), and its execution environment. This helps tailor down the dynamic weaver infrastructure according to specific application requirements. In certain applications such as embedded systems, there is not much variation in terms of the information regarding classes, since the set of classes, and thereby the set of available joinpoints, is usually

known in advance. Thus, when constructing a dynamic weaver for such systems, the feature "***JoinPoints Known***" is selected. This enables to do compile time matching of the aspects to their respective join points for which they will later be registering themselves. In some systems, there might be requirements to apply aspects in specific modules, such as the potential points of interest for system strategies and other cross-cutting concerns. The vast majority of joinpoints in a system is never used by any aspect, as many joinpoints hardly contribute to the application semantics. The execution or control flows of basic library functions, for instance, can be considered as such "low-semantics joinpoints". Thus, in our approach, it is possible to explicitly filter the huge set of available joinpoints to a quite small subset by selecting the feature "***JoinPoints Filtered***", which results in a very efficient system, since unnecessary checks are avoided at each joinpoint.

If even the set of potential aspects is known in advance ("***Aspects Known***"), it is possible to generate such a filter automatically from their pointcut descriptions and thereby, registering only those joinpoints, which are going to be affected by the aspects. Furthermore, if the number of aspects is known in advance, it is possible to fix the size of runtime advice lists associated with each joinpoint, and thereby, avoiding the use of costly dynamic data structures.

In some cases, there is a need to define the order of execution "***AspectsOrder***" of the advices, to resolve the conflicts between different advices, affecting the same joinpoints. The order of activation is supported in static weaving technologies like AspectC++ [19], AspectJ [4]. In certain dynamic weavers 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 "***AspectsOrder***". Moreover, if all the aspects are known in advance "***AspectsKnown***", then the order of advice execution can be defined and resolved statically, saving runtime.

The joinpoint model can also be defined, as per the application requirements, by selecting only those features from "***Supported AOP Features***" which are needed, and same is the case regarding support for different type of advices (***before, after, around***), and for changing the static structure of ("***Introductions***") the program.

In the case of extensible systems, the feature "***ExtensibleSystem***" is needed to be selected. The selection of this feature means that dynamic aspects can not only be applied to the main application, but also to the extension modules. A more detailed description of how this feature enables the dynamic aspects to be woven in distributed environment will be provided in the next section.

The different instances of the weaver are generated completely automatically by the variant management tool by selecting the required features. This truly application-oriented weaver construction drastically reduces the costs (in terms of performance and memory consumption) of the dynamic aspect weaving infrastructure. If the set of effective joinpoints is small, it should even be feasible to implement dynamic aspect weaving as efficient as dynamic class loading.

5 Dynamic Weaver Implementation

Aspect weavers should be able to support both static as well as dynamic weaving, thus combining the advantages of both techniques. Aspects that do not need to be adapted at runtime should be woven statically for performance reasons. In the following subsections, we give an overview of the subset of the features currently supported by our family-based weaver, the weaver architecture, and a discussion of the different variants that can be instantiated from the feature model. The architecture consists of three main modules:

- Weaver Binding
- Run-time monitor
- Dynamic Aspects (shared libraries)

All these modules are completely independent of each other. In this specific construction of the family member, the main aim is to provide low-cost dynamic weaving.

5.1 Weaver Binding

The weaver binding generates information about the joinpoints in the system. This could be a symbol table, generated by the compiler, in the case of binary code manipulation employed as a weaver binding, or a joinpoint information repository, in the case of static weaver employed as a weaver binding etc. The selection of the binding mode is completely dependent on the specific application requirements.

In the current implementation of the different variants of the weaver family, "***Static Weaver***" is used as a weaver binding for code instrumentation. Use of a static weaver as a binding mode helps support both static as well as dynamic aspects. Here, the main idea is that all the potential dynamic joinpoints can be controlled by a static aspect implementation. There are some available static weavers like AspectJ and AspectC++, but again this selection is dependent on the specific application requirements. We are using AspectC++ for this purpose. AspectC++ is an extension to C++, and facilitates to have a dynamic weaver with a very small memory foot print. The code below shows, for example, how we are hooking the execution joinpoints in the base code of some application by making use of pure virtual pointcut in the static aspect.

```
aspect beforeafterInstr{
        pointcut virtual dynamicJPS() = 0;
        public:
              advice dynamicJPS():around(){
                   monitor->BeforeAdviceList(JoinPoint::id ());
                    tjp->proceed();  //method itself is called
                   monitor->AfterAdviceList(JoinPoint::id ());
                }
};
```

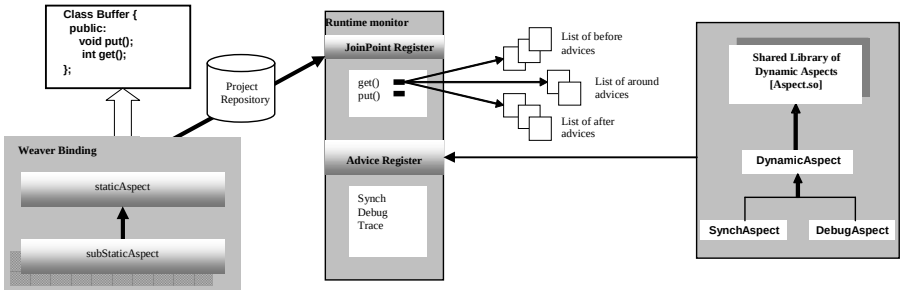


Fig. 3. Dynamic weaver architecture with static aspect as weaver binding mode

```
aspect beforeafterExe:public beforeafterInstr{
        pointcut virtual dynamicJPS() = execution("% ... :: %()" );
};
```

It can be seen from the above code that the around advice is not supported in this specific variant. Thus, by means of derived aspects, joinpoint filtration mechanism is performed. This results in minimal hooks being inserted into the component code and hence, provides us with an efficient and portable dynamic weaver.

5.1.2 Runtime Monitor

The runtime monitor is responsible for coordinating between the aspects (advices) and the component code (e.g. “class Buffer” in figure 3). All the joinpoints and aspects are registered with the runtime monitor.

In our different variants implementations, static AspectC++ weaver generates an XML based file containing information about the joinpoint signatures and their unique ids, which are going to be affected by the dynamic aspects. The joinpoints are registered with the runtime monitor. This information is used by the runtime monitor to create data structures for each potential dynamic joinpoint.

5.1.3 Aspects

We are working to build a weaver family, where the static as well as the dynamic aspects could be described with a single description language. Whenever some advice registers with the runtime monitor, it carries with it information about the joinpoints that it is going to affect. The list of joinpoints registered with the monitor is traversed to find out the joinpoints on which this advice is interested. Three lists of pointers to before, after and around advices are maintained against each affected joinpoint. If there is an around advice registered for some joinpoint, then it means that the around advice is executed instead of the joinpoint.

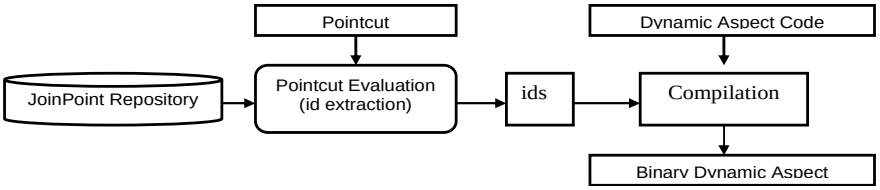


Fig. 4. Compilation process of dynamic aspect

In our current implementation, static aspects are described using AspectC++ language, and the dynamic aspects are simple C++ classes. The dynamic aspects are shared libraries, and so the advices are loaded at runtime. In the case of non-extensible systems, the compilation process of the dynamic aspects is shown in figure 4. The pointcut expression describes the joinpoints of interest by means of string matching and wild cards. The XML file contains the joinpoint signatures and their unique ids. At the compile time, pointcut evaluation is performed by extracting id information from the respective XML file, to find out the ids of the joinpoints that are going to be affected by the dynamic aspects. Thus, at runtime, id matching is performed instead of joinpoint signature matching, which results in an efficient runtime system. Once ids are extracted, they are compiled in combination with the dynamic aspect with a standard ***gcc*** compiler. The object file is converted into a shared library, which can then be loaded and unloaded at runtime into the system by means of "weave" and "unweave" commands from the shell. But in the case of extensible systems where the aspects, already into the system, might be required to affect the modules coming later into the system, this poincut evaluation is completely done at runtime. In our implementation, more than one advice can affect the same joinpoint. Therefore, the feature "AspectOrder" can be selected from the feature model in order to resolve the conflicts between multiple advices. In such variant of weaver, each advice carries with it, its priority number. So when a certain joinpoint is reached, then, these advices are executed as per their priorities or order of execution.

5.2 Dynamic Weaver Implementation in Extensible/Distributed Systems

A dynamic weaver variant to support weaving aspects into the modules loaded later into the running system can be generated from the feature model by selecting the feature "***ExtensibleSystem***.

In such a scenario, in the start, we have the "main application" running, which is affected by a static aspect, as shown in figure 5. This produces information of dynamic joinpoints of the main application in the form of an XML file, named as *0.acp*. The main application, as well as each module, either remote or on the same machine, added later to the system, has its unique runtime monitor object, which registers itself with a monitor registering object called "MonitorRegister". The registration of monitors happens only in the case of extensible systems. In case of non-extensible systems, no such monitor registration is employed, thus, saving resources. Additional modules are loaded into such extensible system at runtime. The same static aspect affects all

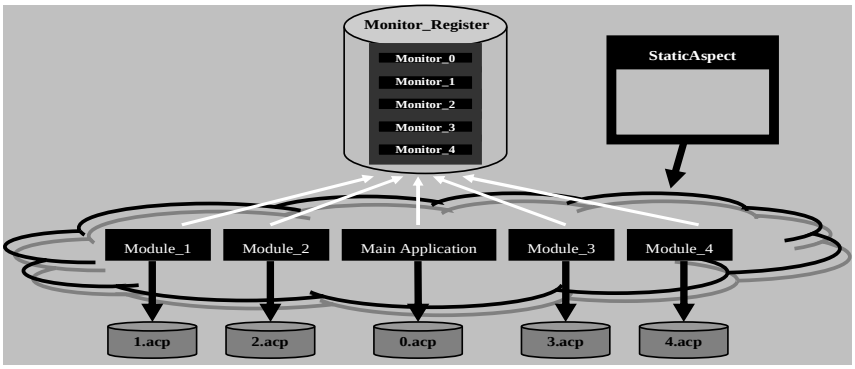


Fig. 5. Dynamic weaver for extensible system

the new modules, and produces their corresponding joinpoint information files (.acp files). We could have used separate static aspects for the instrumentation of these different modules, but this purely depends upon the specific application requirements. In our proposed mechanism, it is possible to generate a customized weaver for each of the modules separately, but this is again a question of specific application requirements. The white indicate each module registering its monitor object with "Monitor-Register". Each of these monitor objects is associated with a specific module, and hence, only registers dynamic joinpoints and aspects for its respective module. Dynamic aspects for extensible systems can specifically be compiled to affect only the main application, one of the modules, multiple modules or some combination of the main application and the modules. While writing the dynamic aspects for different modules, first the monitor object for a specific module is extracted from the monitor repository (MonitorRegister). The joinpoint repositories might be located in different remote locations. The developer has access to the joinpoint repositories for different modules as well as to the main application. This enables him to write an aspect which could affect any joinpoints of interest in the whole system, whether it is any combination of modules or the main application.

6 Results

Dynamic weaving is generally an expensive approach and is not affordable in resource constraint domains like embedded systems. The idea of family-based weavers enables to produce a very optimized and low-cost dynamic weaver, which could be viable for even resource constraint systems. Our approach has advantages over other existing approaches in two ways. First, it allows for the optimization of the runtime support for each application according to its specific requirements in comparison to the generally fixed runtime support offered by other available dynamic weavers. And, secondly, our approach allows for the minimal hooking of the base code to build a weaver which is efficient and highly portable.

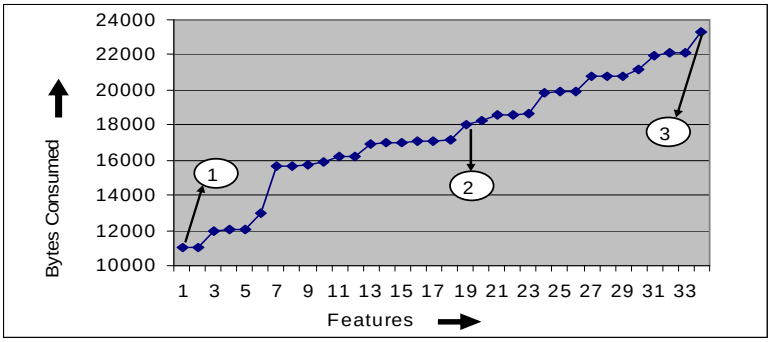
6.1 Cost of Runtime System

Our approach makes it possible to build dynamic aspect weavers with as much runtime support as one application can afford. Each feature in our dynamic weaver feature model has a fixed cost, associated with it in terms of memory consumption. Table 1 shows cost in terms of byte consumption of some of the features from the dynamic weaver feature model. One example of saving resources by making use of before hand knowledge of system can be observed from the different cost of advice features presented in the table. If the number of aspects, going to affect the system, is known in advance, the cost of "***before***", "***after***" and "***around***" advice supported by the runtime system falls around 22%. This is due to the reason that in the case of the feature "***AspectsKnown***" selected in combination with any of the "***before***", "***after***", or "***around***" advice features, each of these advice features is mapped to the implementations that make use of fixed size arrays. The size of array is defined by the user while generating the dynamic weaver from the variant management tool according to the before hand knowledge of the number of the aspects going to affect the system. In the case of the number of aspects not known in advance, the different advice features are mapped to implementations that use costly dynamic structures. This results in a more expensive system in comparison to the one with beforehand knowledge of the number of aspects. Furthermore, the cost of the feature "***AspectsOrder***" is quite considerable (2155 bytes). This feature is needed to be selected only if the different advices coming into the system for the same joinpoint have interdependencies, which is not a very usual case. The weaver becomes considerably expensive if it is to be constructed for extensible systems. The cost of the feature "***ExtensibleSystem***" is the maximum in the whole feature model and stands at 5078 bytes of memory.

Table 1. Feature cost in terms of memory consumption

	Features	**Cost in Bytes**
1	before	1192
2	after	1192
3	around	1331
4	AspectsOrder (order of execution)	2155
5	ExtensibleSystem	5078
6	before (Aspects Known)	934
7	after (Aspects Known)	934
8	around (Aspects Known)	1034

Figure 6 illustrates that as we move towards more support for dynamism in building the dynamic weavers, we pay more in terms of resources. Three different variants of the dynamic weaver family are depicted in the figure along with their memory consumption. It can be seen that the construction of the first variant (1) is the lightest one in terms of memory consumption. The features supported by the first variant (1) are limited as shown in the figure. If more support for dynamism is required like that of the third variant (3), more features have to be selected from the feature model. This



No.	Features Supported (Variants of Dynamic Weaver Family)	Cost in Bytes
1	before, AspectsKnown, call, execution	11045
2	before, after, around, AspectsKnown,, Extensible, call, execution	17990
3	before, after, around, AspectsOrder, Extensible, call, execution	23315

Fig. 6. Cost of different variants of the weaver family

means that the construction becomes more expensive. It can be noted from the graph that there is significant difference in memory consumption between the variant supporting only one type of advice (1) and the other one (3) which is extensible, supports all types of advices and their order of execution. The memory cost of different variants of the dynamic weaver ranges from 11045 bytes to 23315 bytes. This means that the construction of a dynamic weaver with maximum dynamism would consume at least twice as much memory as the one with the lightest construction. The goal, while constructing a dynamic weaver for any application, is to get to some point in the graph in figure 6, where we are not exhausted of the resources and we have as much features added to our weaver construction, to support dynamism, as possible.

6.2 Minimal Hooking Approach

Binary code manipulation approach is cheap in terms of resource consumption, but it is not portable and carries limitations like compiler and architecture-specific implementation, fixed runtime support etc.

Code instrumentation avoids these problems and provides a very portable solution. However, there are certain costs associated with code instrumentation. Firstly, the instrumentation of all the joinpoints results in a large memory overhead. Secondly, it is expensive, because the instrumentation of all the joinpoints result in severe performance overhead because of checks performed, at runtime, at each joinpoint to find out if there is any advice registered. As one experiment with our weaver, we took an open source webserver called “MyServer” [23], implemented in C++, to illustrate this point. We instrumented all the joinpoints of “MyServer” including all

the call and execution joinpoints. There were, in total, 1830 joinpoints instrumented. The code size of the executable increased 7.04 times from 102597 bytes to 722589 bytes. This fully instrumented version of server considerably slowed down in terms of the average response time, and the number of connections handled per second.

However, in practice full instrumentation will rarely be needed. The adaptable features are generally affecting a limited number of joinpoints, and therefore we argue that, in no case, there is a need to instrument all the joinpoints. As an example, consider synchronization in operating system kernels. In a previous paper [21], we described the aspect-oriented implementation of a specific synchronization policy in the PURE operating system [22]. An analysis of the source showed 166 different affected joinpoints spread out over 15 classes. Other synchronisation policies might affect some additional joinpoints of the system later on. Even the union of all joinpoints which might be affected by synchronization policies in the future is still much smaller than the thousand of potential joinpoints which exist in the whole system. Therefore, for the adaptation of synchronization policy, hooking is required only in this relevant subset of joinpoints and not in all the potential joinpoints of the system. Thus, for the adaptation of any global policy, a subset of the joinpoints is first needed to be derived from the union of all the anticipated joinpoints on which a particular policy is expected to affect.

Our approach makes it possible to explicitly filter the huge set of available joinpoints to a quite minimal subset, like the potential points of interest for system strategies and other cross-cutting concerns ("***JoinPoints Filtered***"). This minimal hooking approach enables us to avoid unnecessary memory overhead and performance costs.

7 Conclusion and Future Work

This paper illustrates how the ideas of program family concept are applied to the dynamic aspect weaver domain to build application-specific dynamic weavers to support for the runtime reconfiguration of crosscutting features in the family-based software systems. Most of the concerns in complex software systems that need to be adaptable are crosscutting. AOP is applied for the localization and encapsulation of such crosscutting concerns. The implementations of the features that exhibit crosscutting behaviour are mapped by aspects in the feature models. For the weaving and unweaving of such features at runtime, dynamic weaving is employed. To reconcile our demand on minimal resource usage with (inherently expensive) dynamic weaving, we presented a configurable weaver family. Family-based dynamic weaver is presented as a mandatory feature in the development of a complex family-based adaptable software system. Thus, the software system, as well as the dynamic weaver, is customized according to the specific requirements of any application by selecting only required features from the feature model. Moreover, many of the features, which we offer in the feature model of the dynamic weaver, for example around advice and instrumentation of call joinpoints, are not yet supported by other dynamic weavers in the C/C++ domain. Most of the other existing weavers are targeting specific applications, and so do not have scalability problems yet. 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, a few hundred of Kbytes. Now while doing application specific construction of a dynamic weaver for such systems, 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.

As a future work we intend to extend the AspectC++ compiler to have a single language approach. This would mean that both the static as well as the dynamic aspects could be written with the same AspectC++ language. It will be decided only at the configuration time, whether some aspect has to be static or dynamic, depending purely on the application requirements and resource availability.

References

1. G. Kiczales, et al. Aspect-Oriented Programming, In Proceedings of ECOOP '97, Springer
2. D. Lohmann and O. Spinczyk**:** Architecture-Neutral Operating System Components. WiP Session on SOSP'03, October 19th-22nd, 2003, Bolton Landing NY, USA.
3. J. Bonér: AspectWerkz – Dynamic AOP for Java, Proceeding of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004), March 2004, Lancaster, UK
4. The AspectJ Organization, Aspect-oriented programming for Java, www.aspectj.org.
5. D. C. Schmidt and C. Cleeland, Applying Patterns to Develop Extensible ORB Middleware, IEEE Communications Magazine Special Issue on Design Patterns, April 1999.
6. T. Ledoux, et al. OpenCorba: A Reflective Open Broker, Proceedings of the Second International Adaptive and Reflective Middleware Conference on Meta-Level Architectures and Reflection, 1999
7. F. Kon, et al. Monitoring, Security, and Dynamic Configuration with the Dynamic-TAO Reflective ORB, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing, 2000
8. G. S. Blair, et al. Reflection, Self-Awareness and Self-Healing in OpenORB, Proceedings of the first workshop on Self-healing systems, Nov. 2002, Charleston, South Carolina.
9. P. Greenwood, L. Blair, Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System, Computing Department, Lancaster University, Lancaster, UK
10. D. L. Parnas, Designing Software for Ease of Extension and Contraction, IEEE Transactions on Software Engineering, SE-592:128-138, 1979
11. K. Czarnecki and U. Eisenecker, Generative Programming – Methods, Tools, and Applications. Addison-Wesley, 2000

12. A. Popovici, et al. Just-in-time aspects:efficient dynamic weaving for Java, In Proceedings of the 2nd International Aspect-oriented software development conference (AOSD 2003), March, 2003, Boston
13. pure-systems GmbH, Variant Management with pure::variants, Technical report, http://www.pure-systems.com, 2003
14. R. Pawlak, et al. JAC: A flexible framework for AOP in Java. Reflection 2001
15. S. Aussmann and M. Haupt: Axon – Dynamic AOP through Runtime Inspection and Monitoring. First workshop on advancing the state-of-the-art in Runtime inspection (ASARTI'03), 2003
16. Y. Sato, et al. A Selective, Just-In-Time As-pect Weaver. Proceedings of GPCE'03, 2003, Erfurt, Germany.
17. R. Douence, et al. An expressive aspect language for system applications with Arachne, Proceeding of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), Chicago, Illinois, March 2005
18. S. Almajali and T. Elrad**:** A Dynamic Aspect Oriented C++ Using MOP with Minimal Hook. Proceedings of the 2003 Dynamic Aspect Workshop (DAW04 2003), RIACS Technical Report 04.01, March, 2004, Lancaster, UK.
19. O. Spinczyk, et al. AspectC++: An Aspect-Oriented Extension to C++, In Proceedings of TOOLS Pacific'02, February, 2002, Sydney, Australia.
20. C. Bockisch, et al. Virtual Machine Support for Dynamic Join Points, Proceeding of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004), March 2004, Lancaster, UK
21. D. Mahrenholz, O. Spinczyk, A. Gal, W. S. Preikschat, An Aspect-Orientied Implementation of Interrupt Synchronization in the PURE Operating System Family, Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems , Malaga, Spain, June 11th, 2002, ISBN 84-699-8733-X
22. D. Beuche, et al. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems, Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99), St Malo, France, May, 1999.
23. http://www.myserverproject.net/forum/portal.php
24. M. Engel and B. Freisleben, Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects, Proceeding of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), Chicago, Illinois, March 2005
25. M. Engel and B. Freisleben, Using a Low-Level Virtual Machine to Improve Dynamic Aspect Support in Operating System Kernels, Proceeding of the 4th AOSD Workshop on Aspects, Components and Patterns for Infrastructure Software (ACP4IS 2005), Chicago, Illinois, March 2005
26. C. Zhang and H.A. Jacobson, TinyC^2: Towards building a dynamic weaving aspect language for C, Proceeding of the 2nd AOSD Workshop on Foundations of Aspect-Oriented Languages, Boston, March 2003
27. W. Vanderperren, et al. Adaptive Programming in JAsCo. Proceeding of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), Chicago, Illinois, March 2005