On Adaptable Middleware Product Lines

Wasif Gilani
University of Erlangen-Nuremberg
Erlangen Germany
wasif@informatik.uni-erlangen.de

Nabeel Hasan Naqvi
Siemens CT, SE 5
Erlangen Germany
nabeel.naqvi@siemens.com

Olaf Spinczyk
University of Erlangen-Nuremberg
Erlangen Germany
spinczyk@informatik.uni-erlangen.de

ABSTRACT
Middleware helps to manage the complexity and heterogeneity inherent in distributed systems. Traditional middleware has a monolithic architecture, which makes it difficult to adapt to special requirements such as those present in embedded applications. Middleware for small devices has to cope with a broad range of requirements as well as with the stringent resource constraints. In this paper we propose a family-based approach based on aspect-oriented programming (AOP) for the implementation of middleware product lines which are highly configurable and adaptable.

Such an adaptable middleware is statically configured according to the requirements of the specific distributed application. Furthermore, the middleware is also capable of adapting to the dynamics of the distributed embedded system by dynamically reconfiguring itself during runtime. An efficient dynamic aspect weaver is needed for this kind of adaptability. We also discuss a family of dynamic weavers that complements our study of the family based middleware.

Keywords
Middleware product lines, Aspect-oriented programming, adaptable middleware, dynamic reconfiguration, dynamic weaving

1. INTRODUCTION
Conventional middleware e.g. CORBA [1], COM [2], RMI[3] etc. focus on masking out the problems of heterogeneity to facilitate the development of distributed systems. These middleware are designed and built to provide a wide feature set to suit the needs of multiple problem domains. The extra features not used by the application contribute to unnecessary code size and configuration complexity. It is becoming increasingly difficult to achieve a high level of adaptability and configurability of these middleware as their functionality matures, due to the limitation of software decomposition methods. On the other hand the development of software in distributed embedded systems, having real-time requirements, needs highly specialized tools and techniques. Such specializations make it difficult to adapt traditional distributed embedded software to meet functional or quality of service requirements, hardware/software technology innovations or emerging market opportunities. The conventional middleware, therefore, are not suited for use in such distributed embedded environments. Middleware for embedded and deeply embedded devices have to scale with a very broad variety of requirements, pertaining both to the hardware as well as the software level. A system of networked embedded devices exhibits strong dynamics and massive heterogeneity. Different hardware architectures and configurations have to be supported and at the same time the middleware has to take into account the stringent resource constraints of the embedded systems. Different applications typically have different requirements on the services and strategies implemented by the underlying middleware. An example of such system dynamics can be observed in the field of mobile communications. Mobile communication devices comprise a truly heterogeneous system. These mobile devices are based on different hardware and software platform and may make use of different communication media. The requirements of the mobile devices may change with their frequent relocation and strong fluctuation of bandwidth. The limited battery power presents another challenge in the design of mobile communication systems. Some features of the system may have to be readjusted to take into account the change in available battery life [9].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
3rd Workshop on Adaptive and Reflective Middleware Toronto, Canada
Copyright 2004 ACM 1-58113-949-7...$5.00.

2. MIDDLEWARE PRODUCT LINES
Typical middleware architectures struggle between providing generality and specialization. On one hand the middleware needs to support many application domains by providing a large set of features. On the other hand the middleware needs to provide optimization to support special runtime requirements. Such conflicting requirements result in multiple specifications and different implementations. The cost of maintaining the code base is, as a result, much higher.

The above scenario necessitates the need for a customizable middleware, which can be adapted to suit exactly the needs of an application. Several approaches have been adopted to achieve this kind of customizability of the middleware. The ADAPTIVE Communication Environment (ACE) ORB (TAO) [4, 8], for example, tries to provide customization by using strategy patterns in several features. 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.

ZEN [25] is another interesting approach which is based on real time java [26] and also makes use of design patterns. This ap-

proach allows both static as well as dynamic configuration. Other approaches such as those adopted by DynamicTAO [5], OpenORB [27] and Open CORBA [6] suggest the use of reflection and component frameworks. In some of these approaches [27], the middleware 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 middleware. The drawback of these techniques is that these have rather large memory requirements and these also incur performance overhead. This renders the above-mentioned techniques less suitable for use in deeply embedded systems.

Thus, it is not possible to build a middleware which 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 middleware so that it provides only the services needed by any particular application. This leads to a *family-based* [22] or *product-line* approach, where the variability and commonality among middleware family members is expressed by *feature models* [20]. Special tools are used to extract and statically configure the concrete middleware based on an application-specific feature selection. The process of selecting features and setting defaults to generate a member of the family of products is known as application engineering. A crucial point is the mapping of all selectable and configurable features to their corresponding, well encapsulated implementation components. The encapsulation of non-functional properties is often limited, due to their crosscutting character. This makes it almost impossible to implement them as independent encapsulated entities and thereby restricts variability and granularity. The solution to the above problem lies in carrying out the design process in which requirements orthogonal to the fundamental functionality of the middleware are separated. *Aspect-oriented programming (AOP)* [21] has provided us with the solution to deal with these non-functional properties or crosscutting concerns.

3. ASPECTS IN MIDDLEWARES

Non-functional requirements pertain to requirements that are not included directly in the functionality of the middleware. These rather express additional characteristics that the middleware should have. Examples are fundamental system policies such as synchronization, real-time capability, quality of service (QoS) and security.

A very important example of a crosscutting concern is the quality of service (QoS). The primary goal of the QoS is to provide priority including dedicated bandwidth, jitter, latency and improved loss characteristics. The middleware in distributed embedded systems must be capable of configuring the QoS statically as well as dynamically.

Aspect-oriented programming (AOP) [21] has proved to be an effective way of localizing and encapsulating such cross cutting concerns. The presence of crosscutting concerns means that a single dimension of functional decomposition is not sufficient for a modular design of the system. Crosscutting concerns cannot be encapsulated in a single function, class or module. The so-called aspect code tangles with the functional component code that fits into the functional decomposition scheme.

AOP encapsulates the implementation of crosscutting concerns in 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. Aspect weaving can be done both at compile time, called static weaving, as well as at runtime called dynamic weaving.

Appropriate use of AOP in the development of middleware product lines results in a higher variability and granularity of the selectable middleware features. The resulting domain model may even allow configuration of fundamental architectural properties.

4. STATIC CONFIGURATION

Static configuration is the stage where the user actually selects from the features presented in the feature model, and set defaults according to the needs of the application. This process is called application engineering and this is the counterpart of domain engineering [20]. Application engineering is the process of actually generating the member of the family of products developed earlier as a result of the domain engineering process. A product line is obtained by carrying out a feature-oriented domain analysis of the specified domain. The core functionality consists of a subset of features common to all configurations. This results in a smaller memory footprint and a simpler implementation, which is vital for embedded systems. Aspects representing the crosscutting requirements can then be superimposed onto the primary functionality in an additive manner without altering the existing architecture.

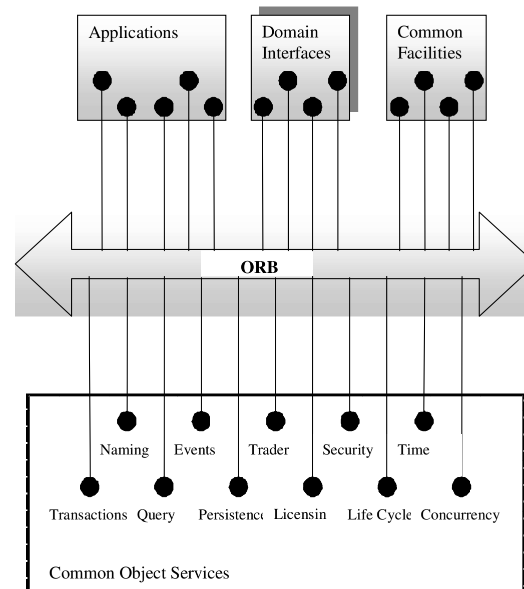

Figure 1. Typical implementation of ORB

We consider the CORBA object request broker (ORB) as a case study of customization. The process of static customization applies to both the basic functionality as well as the orthogonal non-functional requirements. The typical implementation of an ORB contains many common services as shown in Figure 1. An application running in a specific domain context needs only a

subset of these features. The various ORB services may themselves be configured to provide the variant of the ORB service appropriate for the specific application context. One example is the middleware employed in financial systems. Financial systems performing secure transactions need security and transaction processing features. Online information systems such as online catalogues do not have such requirements. Another example is that of embedded systems having realtime and availability requirements such robotics, avionics etc. ORBs for embedded systems therefore have to provide services for fault tolerance and realtime response. By statically configuring the ORB, therefore we include only the features needed by a specific application. This results in an efficient ORB with a low memory footprint. Another example of customization of the core ORB functionality is its data type definition system. An application running in the context of a middleware may not be making use of all of the data types provided by the type definition system of the ORB [23]. Tailoring the type definition system down to only the data types needed by the application would also result in lower memory footprint.

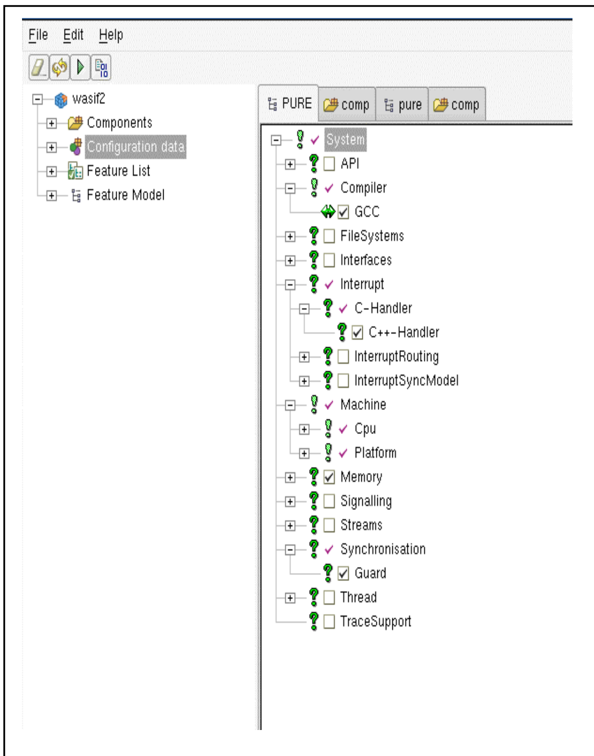

Figure 2. Consulat tool for statically configuring the PURE family of operating system

A variant management system is needed to specify default dependencies in the feature model to prevent the combinatorial explosion of the variants. An example of such a tool is the PURE::CONSUL[12], as shown in figure 2, which has been successfully employed for the variant management of the PURE operating system family [24] for embedded systems. PURE::CONSUL provides a graphical user interface in which is displayed a hierarchical representation of the feature model of the product family. The user selects feature nodes, which are mapped onto implementation components. This information is fed to generators, which output and build the final product. The result of static configuration is a product, which contains only those features, which are needed by the application.

5. DYNAMIC RECONFIGURATION

The motivation for our work on dynamic adaptation is to provide a better, i.e. more resource efficient, middleware support for applications in a dynamically changing environment than it could be provided by a one-fits-all solution. The general idea behind our approach is that as much processing as possible should be done at compile time, because of the limited runtime resources in the field of embedded systems. Once statically configured, the middleware may be subject to the changing requirements during runtime. This is especially true in distributed embedded systems, which exhibit strong dynamics. We argue that a certain mechanism should be available to enable the pushing in and pulling out of the services at runtime. This enables middleware to keep on a node only those services that are required by an application. In order to influence the runtime behavior of the underlying middleware, the application needs to gain access information about its execution context. Some approaches e.g. DynamicTAO and OpenCORBA suggest the use of reflection and metadata for this purpose. These approaches are not suitable for embedded applications, which operate under strict resource constraints. Therefore we need some other efficient mechanism for efficiently adding and removing the services from the system.

Dynamic reconfiguration of the middleware is based on policies that describe the non-functional requirements specific to the application. Examples are QoS requirements as well as common services such as error logging, synchronization, tracing etc. Besides dynamic loading and unloading of usual modules, dynamic reconfiguration requires dynamic weaving/unweaving of aspects. A dynamic weaver does the job of weaving of dynamic aspects and, hence, is a vital part of the adaptable middleware.

Considering our case study of the ORB, we note that dynamic adaptability is achieved mainly by influencing the method invocation semantics. Following are a few of the possibilities of intercepting the method invocation in an ORB: marshalling/unmarshalling, portable interceptors and dynamic invocation interface (DII) [7]. Marshalling and un-marshalling is the process of packaging/un-packaging parameters of the method invocation. The same is also done for the result values returned. During marshalling objects can be replaced. Furthermore, marshalling can be extended to perform compression and encryption. Portable interceptors are hooks into the ORB, which allow interception of the ORB services at various stages of the request process. Portable interceptors thus allow plugging in of additional ORB functionality such as transaction support and security. Through dynamic invocation interface (DII), invocation to an interface can be composed at runtime without prior knowledge of the interface definitions. DII therefore also provides an interception point for reflecting on invocation on the server objects.

The above approaches make use of the metadata information, which makes these approaches impracticable for dynamic adapta-

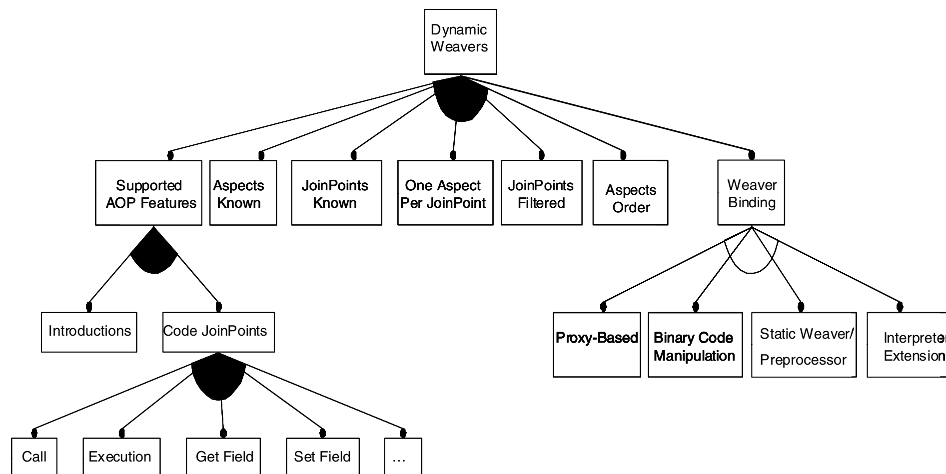

Figure 3. Feature model for dynamic aspect weavers

tion in embedded systems operating with restricted memories. In Section 6 we will be discussing different existing approaches for dynamic weaving, and reasoning about their unsuitability in the domain of embedded systems. Then we will present our approach of generating application-specific dynamic weavers from a family of dynamic weavers, based on the feature selection of the middleware itself. This ensures smaller footprint and better performance of the dynamic weaver.

6. DYNAMIC WEAVERS FOR ASPECTS

Dynamic weaving is always an expensive option regarding runtime overhead, and so is not generally acceptable in the domain of embedded systems. However, for the construction of adaptable middleware, the dynamic weaver is one of the fundamental features in the feature hierarchy, and needs to be selected for the adaptation of global policies as per runtime requirements.

Several approaches have been proposed by the AOSD community for supporting dynamic weaving. Most of these use reflection and metadata information. Java specific approaches use interception mechanisms (proxy-based), byte code manipulation or virtual machine extensions [13, 16, 17, 19]. These approaches are not suitable for embedded systems domain as they use lot of memory. So far only few approaches have been proposed for the C/C++ domain [14, 15]. The approaches in C/C++ basically follow two general implementation techniques. Firstly, the aspects are woven at runtime by on-demand insertion of jump statements to the aspect code into the machine code at all affected joinpoint positions. This technique is followed by the MicroDyner project [15]. In the second approach, aspects are woven by registering them against a runtime registration system. The runtime registration system manages lists of all registered aspects and all available joinpoints and thereby performs the binding of aspects to joinpoints. 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. This technique is followed by the DAO C++ project [14].

Even the approaches proposed for the C/C++ domain are not acceptable for the embedded domains, as they are quite expensive at runtime. This is especially true for the DAO C++ approach, where the runtime system has to be called at each potential joinpoint, regardless if there is an aspect registered for this joinpoint or not. The MicroDyner approach avoids these costs by on demand weaving in the machine code, which should perform much better at runtime. However, the machine code itself has to be prepared to support dynamic weaving. For the joinpoints to be visible at machine code level, the compiler must not optimize or inline any part of the code. And finally, this is a machine- and compiler-specific technique and therefore not practicable for the broad variety of hardware platforms in the domain of (deeply) embedded systems.

The dynamic weaver is an important means by which features implemented as aspects can be adapted at runtime by weaving and unweaving. But most of the existing weavers are not suitable for the domain of embedded devices because of either their excessive use of memory and runtime or compiler specific solutions like MicroDyner. In the later section we propose using a family-based approach of dynamic weavers to extract application-specific dynamic weavers, which are suitable for applications running in a distributed embedded environment.

6.1 Family-based Dynamic Weavers

To overcome the deficiencies in existing dynamic weaver approaches, we have proposed a family-based approach of constructing application-specific dynamic weavers from a family of weavers [11]. We follow the approach that less demanding applications should not be forced to pay for the resources consumed by unneeded features. Thus, not only the middleware is scalable according to available resources, but also the dynamic weaver. Following this approach, the weaver construction is parameterized with specific environment constraints, which are defined by a feature selection. These weavers are based on the technique of runtime aspect registration, but can be tailored down according to the specific application requirements. All dynamic aspect weaver implementations, we have examined, provide a fixed set of AOP features that can be applied at any potential joinpoint. For example, DAO C++ supports only a limited joinpoint model, namely the execution of before and after advices, for a joinpoint. Moreover, there is no joinpoint filtration mechanism and any loadable aspect is able to affect any joinpoint. This means that calls are generated

from every joinpoint to the runtime system and, thus, result in large overhead. Even functions that will never be affected by aspects are slowed down and require more memory space.

We have developed a feature model following the family-based approach as shown in figure 3. The dynamic weavers are constructed from this feature model by selecting only those features, which are needed by the application. Thus, not only the middleware construction is application-oriented, but also the dynamic weaver, which is itself an optional feature in the feature model of the middleware. Thus, a truly application-oriented system evolves. This system is quipped with a resource sensitive dynamic weaver to enable dynamic adaptation of the features of middleware, which have runtime behaviour, and are realized as dynamic aspects. In the construction of the dynamic weaver, 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. For example, the selection of binding mode ("Weaver binding") feature is selected purely depending on the particular requirements and available resources of a certain application. In certain applications such as embedded systems, there is not much variation in terms of the information regarding classes. Thus the set of classes, and thereby the set of available joinpoints, is usually known in advance ("JoinPoints Known"). Hence, it is possible to do compile time matching of aspects to their respective join points for which they will later be registering themselves. Also in most of the existing approaches there is insertion of hooks to all the joinpoints, and so even if no aspect is registered for some joinpoint, the runtime system is called to check for the aspects and results in unnecessary runtime overhead. In our approach, it is possible to explicitly filter the huge set of available joinpoints to a quite small subset ("JoinPoints Filtered"), which results in a very efficient system. 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. Furthermore, in some cases it is known how many aspects are going to affect the system ("Aspects Known"). Thus 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. Moreover only those joinpoints, which are going to be affected by these aspects, are registered. This results in more efficient system. Also when the feature "Aspects Known" is selected then it automatically means that the feature "Aspect Order" is needed to be selected as well. Thus if all the aspects are known in advance, then the order of aspect 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 the same is the case regarding changing the static structure of ("Introductions") the program.

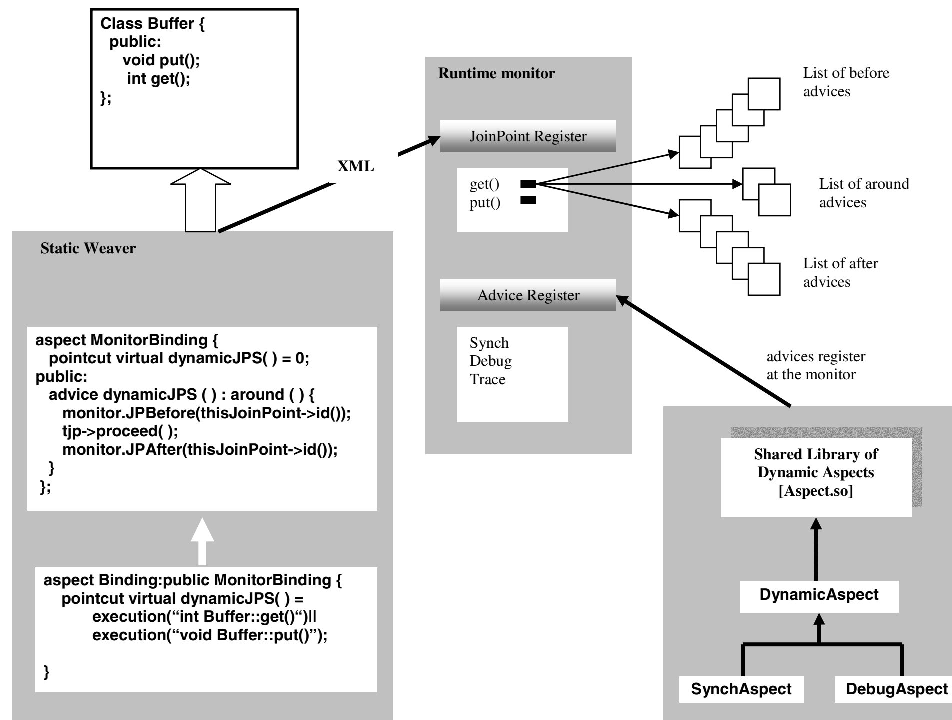

Figure 4. Dynamic weaver architecture

The main idea behind this approach is to be capable of building *low cost dynamic weavers* by incorporating before hand knowledge about the system (domain analysis), and its execution environment, to tailor down the dynamic weaver infrastructure. This truly application-oriented weaver construction drastically reduces the costs (in terms of performance and memory consumption) of dynamic aspect loading. If the set of effective joinpoints is small, it should even be feasible to implement dynamic aspect loading as efficient as dynamic class loading.

6.2 Generating a Dynamic Weaver

In the specific dynamic weaver construction from the weavers family, as shown in figure 4, we are making use of "a Static Weaver" (AspectC++ [18]) as a binding mode. We could have used some other available static weaver like AspectJ [10], but again approaches based on Java are not feasible for embedded

systems domain because of memory constraints. The use of AspectC++ results in an efficient solution regarding the development of an application-specific and an adaptable middleware. This construction consists of three main modules:

- Static Weaver (mandatory feature)
- Run-time monitor
- Dynamic Aspects (shared library)

All these modules are completely independent of each other. It is possible to select any other binding mode according to specific application requirements. In this specific construction, the main aim is to have the ability of providing *low cost dynamic weaving*. AspectC++ is an extension to C++, and so facilitates to have a dynamic weaver with very small foot print. It is used originally to describe static aspects, and is a general-purpose aspect-oriented extension of C++, which follows an AspectJ like approach of AOP. It is implemented as a C++ preprocessor, based on a source code transformation system that transforms aspects into C++ code. This generated C++ code is then compiled using a conventional C++ compiler. Use of a static weaver as a binding mode helps support both static as well as dynamic aspects. Moreover in this approach, it is assumed that more than one aspect can affect the same joinpoint. Here the main idea is that all the potential dynamic joinpoints can be described by a static aspect implementation. In this approach, only a limited number of hooks are inserted into the component code for the joinpoints, which are going to be affected by aspects. The runtime monitor is basically responsible for coordinating between the aspects (advices) and the component code (“class Buffer”). One important variation in our approach from other existing approaches is that we are not changing the semantics of dynamic AOP from static AOP. We believe that an aspect should behave in the same way in dynamic weaving as it does in the static weaving. A dynamic aspect should be able to contain multiple advices for different joinpoints like the static aspect. It should not be limited to contain just a “before” and an “after” advice, as is being promoted by most of the existing dynamic weaving approaches. In this construction of a dynamic weaver, a dynamic aspect is a simple C++ class. It can contain multiple “before”, “after” and “around” advices for different joinpoints which are simple C++ methods. Moreover, each advice registers itself with the runtime monitor. The static AspectC++ weaver generates an XML-based file containing all the joinpoints, which are going to be affected by the aspects. These joinpoints are also registered with the runtime monitor. This information is used by the runtime monitor to create data structures for each potential dynamic joinpoint. In the case of joinpoints not known in advance, this XML file can be converted to a binary format to make the processing efficient. The set of affected potential joinpoints are controlled from the static aspect. This joinpoint set, as well as features selected from the feature model, can be tuned according to advance knowledge, to reduce the overhead, which is related to the weaving infrastructure. The dynamic aspects are shared libraries and so the advices are loaded at runtime. Whenever some advice registers itself with the monitor, the list of registered joinpoints is traversed to find out, on which joinpoint this advice is interested. Three lists of pointers to before, after and around advices are maintained against each affected joinpoint as shown in figure 4. If there is an around advice registered for some joinpoint, then it means that the control of execution is never returned to the actual joinpoint. In case, when there is no around advice registered for a joinpoint, before and after advices of each joinpoint are invoked by the runtime monitor, whenever some joinpoint is reached by a thread of control.

In the current implementation, static aspects are written using AspectC++, and the dynamic aspects using simple C++ language. We are working to extend the AspectC++ compiler to have a single language approach. Thus, it would be possible to write both static as well as dynamic aspects 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.

7. CONCLUSIONS

In this paper we have presented our ideas of building application-specific adaptable middleware by making use of the program family concept and dynamic weaving. Adaptability and configurability requires a very high level of modularity in the middleware architecture. We have shown that the family-based approach together with AOP can be used to achieve this kind of modularity. A middleware family is obtained by carrying out a feature-oriented domain analysis of the middleware domain. The structure of our family implementation motivates the need for dynamic weaving where each feature is implemented as a module. The features which exhibit crosscutting behaviour are realised as aspects. Thus, for the weaving and unweaving of features at runtime, which are implemented as aspects, dynamic weaving needs to be supported. To reconcile our demand on minimal resource usage with (inherently expensive) dynamic weaving, we presented the idea of a configurable dynamic weaver family that makes use of *a priori* knowledge about possible system changes. Dynamic weaving is viewed as an optional feature in the feature model of the middleware. This feature is clearly expensive and so is selected only when, for certain applications, there is a need for dynamic adaptation of middleware at runtime. The program family approach helps extracting an application-specific dynamic weaver from the available family members.

We are in the process of carrying out the domain engineering of middleware. Our aim is to develop a generative domain model for a family of adaptable middleware. Moreover, we are implementing all the features for the family of dynamic weavers. Major parts of this family have already been implemented. We are also working on to make AspectC++ as one standard language to define both static as well as dynamic aspects. We will be performing some measurements to prove our assumption that resource consumption can be scaled down as per the requirements on dynamism. Thus, even in resource constrained domains, there will be a possibility to build and afford dynamic weavers which would be able to support at least some level of dynamic weaving.

8. REFERENCES

[1] http://www.corba.org

[2] http://www.microsoft.com/com/

[3] http://java.sun.com/products/jdk/rmi/

[4] http://www.cs.wustl.edu/~schmidt/TAO.html

[5] http://choices.cs.uiuc.edu/2k/dynamicTAO/

[6] T. Ledoux, W. Cazzola and F. Rivard, *OpenCorba: A Reflective Open Broker*, Proceedings of the Second International

Conference on Meta-Level Architectures and Reflection, 1999

[7] C. Zhang, H. Jacobson, *Quantifying Aspects in Middleware Platforms,* Proceedings of the 2nd international conference on Aspect-oriented software development table of contents, Boston, Massachusetts, 2003

[8] F. Hunleth, R. Cytron, and C. Gill, *Building Customizable Middleware Using Aspect Oriented Programming*, OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa, Florida USA, October 2001.

[9] L. Carpa, W. Emmerich and C. Masolo , *Middleware for Mobile Computing: Awareness vs. Transparency,* In Proc. of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII). Schloss Elmau, Germany. May 2001

[10] The AspectJ Organization, *Aspect-oriented programming for Java*, www.aspectj.org.

[11] W. Gilani and O. Spinczyk , *A Family of Dynamic Weavers*,. Proceedings of the 2003 Dynamic Aspect Workshop (DAW04 2003), RIACS Technical Report 04.01, March, 2004, Lancaster, UK.

[12] Technical White Paper, *Variant Management with Pure::Consul*,http://web.pure-systems.com/fileadmin/downloads/pureconsul_en_03.pdf

[13] P. Netinant, C. A. Constantinides, T. Elrad, and M. E. Fayad**:** *Supporting the Design of Adaptable Operating Systems Using Aspect-Oriented Frameworks*, Proceedings of the International Conference of Parallel and Distributed Processing Techniques and Applications (PDPTA'00), pp. 271-278, June 2000, Las Vegas, NV, USA.

[14] 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.

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

[16] Y. Sato, S. Chiba, and M. Tatsubori: A Selective, Just-In-Time Aspect Weaver. Proceedings of GPCE'03, 2003, Erfurt, Germany.

[17] 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

[18] O. Spinczyk, A. Gal, and W. Schröder-Preikschat: *AspectC++: An Aspect-Oriented Extension to C++*,. In Proceedings of TOOLS Pacific'02, February, 2002, Sydney, Australia.

[19] A. Popovici, T. Gross, and G. Alonso, *Dynamic Weaving for Aspect Oriented Programming*. Proceedings of AOSD'02, April 2002, Enschede, The Netherlands.

[20] K. Czarnecki and U. Eisenecker, *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, 2000

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, Aspect-Oriented Programming, In Proceedings of ECOOP '97, Springer

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

[23] S. Apel, *Towards a Flexible Tailor-Made Middleware for Mobile Distributed Information Systems*, Technical Report of the 20th British National Conference on Databases (BNCOD20), pages 33-40. University of Coventry, School of Mathematical and Information Sciences, July 2003. Presented at the PhD Forum of the BNCOD20.

[24] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk, *The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems*. Proceedings of ISORC'99, May 1999, St Malo, France.

[25] R. Klefstad, D.C. Schmidt, and C.O'Ryan, *Towards Highly Configurable Real-time Object Request Brokers*, Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing , April 2002, Washington D.C.

[26] G. Bollella, B. Brosgol, P. Dibble, J. Gosling, S. Furr, D. Hardin, M. Turnbull, and R. Belliardi, *The Real-Time Specification for Java*, 2000, Addison-Wesley.

[27] G. S. Blair, G. Coulsan, L. Blair, H. Duran-Limon, P. Grace, N. Parlavantzas, and R. Moreira, *Reflection, Self-Awareness and Self-Healing in OpenORB*, Proceedings of the first workshop on Self-healing systems, Nov. 2002, Charleston, South Carolina.