Towards Just-in-time Middleware Architectures*

Charles Zhang, Dapeng Gao and Hans-Arno Jacobsen
Department of Electrical and Computer Engineering
and Department of Computer Science
University of Toronto
{czhang,gilbert,jacobsen}@eecg.toronto.edu

ABSTRACT

Middleware becomes increasingly important in building distributed applications. Today, conventional middleware systems are designed, implemented, and packaged prior to their applications. We argue that with this middleware construction paradigm it is often difficult to meet the challenges imposed by application specific customization requirements. We propose to reverse this paradigm by automatically synthesizing middleware structures as the result of reasoning about the distribution needs of the user application of middleware. We term this type of post-postulated middleware *Just-in-time middleware* (*JiM*). In this paper, we present our initial design and present an evaluation of the *JiM* paradigm through Abacus, a CORBA middleware implementation based on the aspect oriented refactoring of an industrial strength object request broker. In addition, we present Arachne, the Abacus synthesizer, which integrates source analysis, feature inference, and implementation synthesis. Our evaluations show that, through automatic synthesis alone, Abacus is able to support diversified application domains with very flexible architectural compositions and versatile resource requirements as compared to conventional pre-postulated approaches.

Keywords

Aspect Oriented Middleware, Middleware Architecture

1. INTRODUCTION

The purpose of middleware such as CORBA[1], .NET Remoting[2], and Web Services[3] is to provide uniform and native representations of remote services for distributed applications to avoid dealing with the high degree of heterogeneity in hardware platforms, operating systems, communication protocols, programming languages, application semantics, and many others. Traditional middleware architectures are often criticized as monolithic or coarse-grained. This high degree of inflexibility has called for newer paradigms to build versatile middleware architectures which aptly adapt to ever changing external execution environments as well as requirements of features desired by the user applications of middleware.

Many new architectural paradigms have been proposed, successfully exploiting techniques such as reflection [2, 4, 11], component frameworks [10], and aspects [14, 8]. A common characteristic of these solutions, as well as traditional approaches, is that middleware functionality is independently conceived and packaged as either active services or framework libraries according to which applications are developed. We term these middleware architectures *pre-postulated architectures* with respect to the application development time. The pre-postulated solutions have the following limitations:

1.***Impedance mismatch*** – Existing middleware architectures are typically designed according to specific and yet coarse classifications of application domains such as enterprise platforms, resource-constrained environments, safty-critical systems, and many others. However, user applications might not be appropriately categorized in the same way. Consequently, pre-postulated architectures might be either functionally redundant (overfit), or insufficient (underfit), or incomplete (partial fit). Generally speaking, pre-postulation is infeasible in providing a tailored architecture for a specific user application.

2.***Application obliviousness*** – Pre-postulated architectures cannot take full advantage of the rich information embodied in user applications regarding their required middleware features such as data types, invocations styles, programming styles, and additional middleware services. It is very difficult to pre-design a customizable and adaptive system which takes the full diversity of user applications into consideration.

3.***Domain limitations*** – Pre-postulated middleware solutions typically target specific application domains and are often problematic in supporting applications in other domains. This is because a large number of extrinsic, i.e., domain-specific, properties are implemented in a non-modular

*In Fourth International Conference on Aspect-Oriented Software Development, Chicago, USA, March 14-18, 2005

[1]Common Object Request Broker Architecture. URL: `www.omg.org`

[2]Microsoft .NET Remoting. URL: `http://msdn.microsoft.com/library/en-us/dndotnet/html/hawkremoting.asp`

[3]W3C Web Services Activity URL: `http://www.w3.org/2002/ws/`

and convoluted fashion[4]. More importantly, it is difficult to pre-postutate all of the variations of the deployment environments in each and every user application scenario. Therefore, conventional middleware conceived for one application domain typically cannot readily support a different domain without considerable re-implementations.

4.***Excessive configurability*** – Middleware composed of fine granularity have greatly alleviated the difficulty of customization. However, in pre-postulation approaches, software configuration is independent of the middleware architecture and left either minimal in terms of application-specific tailoring or primarily as the user's responsibility. Configurability can be a daunting task because the number of configuration possibilities grows exponentially as several authors have shown in [17, 8].

To solve the afore-mentioned problems, we call for a paradigm shift in the construction of middleware. We propose a reverse design paradigm in which the constituents of middleware implementations are not pre-postulated *for* user applications but post-postulated *from* the knowledge embedded in the user application or the specific target application domain. Borrowing metaphorically from the terminology Just-in-time (JIT) compilation in the Java world where the machine-specific code is generated only during runtime, we term the paradigm *Just-in-time Middleware* (*JiM*), where the executable middleware is only generated at the most appropriate time. Intuitively, the "appropriateness" gets better as the *time* gets closer to the runtime of the application. Currently, we are primarily concerned with the *development time* and defer "later times" till further research.

The *JiM* approach is based on the observation that individual user applications or application domains often do not require all of the middleware features available in standard platforms. This feature redundancy in the final middleware deployed can best be avoided if the distributed computing intent of the user application can be externalized and exploited to drive the composition of the middleware. In *JiM*, we propose the following general stages to achieve this goal: gathering functional requirements from both the user application and environmental constraints (*Acquisition*), reasoning about the final composition of middleware (*Targeting*), and generating the correct middleware as well as the associated test cases (*Synthesis*). The most important premise of enabling this process is a very high degree of modularity and configuration granularity. Traditional middleware architectures, as limited by their hierarchical structures, suffer from the incapability of modularizing crosscutting concerns. This often hinders the separation and the modularization of generic middleware logic in face of domain specific properties as pointed out by Harrison and Ossher [7]. Hence, we believe that aspect oriented programming [9] is one of the enabling technologies for achieving *JiM* architectures. Further, the horizontal decomposition principles [17] serve as effective guidance for implementing *JiM*, as our experience has shown.

In this paper, we make the following contributions:

1. We present the key ingredients of the *JiM* paradigm: *acquisition*, *targeting*, including both *inference* and *verification*, and *synthesis*. More specifically, we propose a feature acquisition process bootstrapped by both remote interface declarations and the user application program source. This process is directed by rule-based dependency and constraint specifications. We propose a code naming schema to organize the code space of aspects in order to facilitate fast and correct synthesis of middleware implementations.

2. We present Abacus, our prototype of *JiM* based on the aspect oriented refactoring of ORBacus[5], an industrial strength CORBA implementation, Abacus is capable of automatically synthesizing specific middleware implementations to manage distribution concerns for applications in different execution environments from embedded to desktop and to enterprise platforms. This synthesis process is facilitated by our aspect-aware IDL compiler and the Arachne synthesizer. We also describe an inference algorithm and motivate the guarantees the algorithm provides for the final middleware composition – correctness and minimalism.

3. We present a thorough evaluation of Abacus through both randomized feature selections and an experimental cross-domain application: an ubiquitous messenger. We measure and report both the static and the dynamic versatility exhibited by Abacus in transforming itself to dramatically different execution environments including cell-phones, desktop machines and enterprise computing settings.

The organization of the paper is as follows: Section 2 briefly introduce middleware and the horizontal decomposition (HD) principles. Although the focus of this paper is on customization, we use the HD principles to enable the architectural flexibility required by *JiM*. In Section 3, we present an abstract discussion of key ingredients for achieving a *JiM* architecture. In Section 4, we present Abacus, a *JiM* implementation. The evaluation of Abacus is presented in Section 5.

2. BACKGROUND

Middleware: We define middleware as a set of services that facilitate the development and the deployment of distributed systems in a heterogeneous networking environment. In the context of this paper, we further narrow this definition to be the software substrate which enables transparent remote invocations of services. The design requirements for middleware are still very complex because they cover two orthogonal middleware characteristics, identity coupling and temporal coupling,among applications requesting services (clients) and ones providing computing services (servers). Identity coupling characterizes how much clients and servers know about each other, and temporal coupling characterizes the degree of synchrony of the message exchange between them. Traditional RPC-based middleware, such as DCOM, CORBA, and Java RMI, exhibits strong identity and temporal coupling. Clients and servers of publish/subscribe middleware, on the other hand, do not know about each others' identities and do not synchronize when communicating either.

Horizontal decomposition: Horizontal decomposition (HD) is a set of principles we have proposed in [17] to guide the aspect oriented design and implementation of complex systems such as middleware. HD principles play the fundamental role in achieving a very high degree of modularity and enabling the *JiM* paradigm because they effectively address the feature convolution problem prevalent in legacy middleware architectures. We use "convolution" to denote

[4]Please see section 2 for the definition of convolution

[5]ORBacus. URL:`http://www.orbacus.com/support/new_site/index.jsp`

large-scale N-by-N interactions among orthogonal features in vertical architectures, those built using hierarchical modules [6]. We perceive that the implementation of an aspect consists of both its functional implementation and its possible interactions with every other aspect. HD promotes a two dimensional architecture in which the vertical architecture implements a minimum set of essential functionality of the application, and the horizontal architecture captures crosscutting concerns including both functional and non-functional features [13]. The horizontal features are decoupled from each other and each can be independently "woven" into the vertical architecture. We term this fashion of architecture "*super-impositional architecture*" [17].

3. JUST-IN-TIME CUSTOMIZATION

"The program is obsolete by the time it is done" [3]. Targeting a very diversified application domain, the design of middleware becomes obsolete even faster. We think that one way of overcoming this "curse" is to postpone at least part, if not all, of the composition of middleware, or software in general, as close to its execution time as possible. We think this is possible because, with the help of higher-degree modularization techniques, it is possible to separate the minimum common functionality of middleware implementations across different domains and to construct extrinsic properties as modules. These modules are constructed in a disciplined way so that they can be combined with minimum constraints. The core-based super-impositional architecture advocated with the horizontal decomposition principles is one way of achieving this goal.

The spirit of Just-in-time customization is to let the distribution intentions of the user applications drive the final composition of middleware. This is contrary to traditional construction methods in which the design and the implementation of middleware are conducted prior to that of the application. With certain assumptions, our current research effort in *JiM* focuses on facilitating the composition of middleware after the user application is developed. The following sections discuss essential *JiM* concepts as well as our general assumptions. We purposely present the discussion in an abstract manner rather than in implementational details. In Section 4, we incarnate these concepts through our concrete implementation which serve as examples for the abstract discussion.

3.1 General assumptions

The complete realization and evaluation of *JiM* is a complex task. However, we are able to test the viability of this approach under the following assumptions:

1. The *JiM* paradigm is based on the existence of a minimum set of functionalities and their associated structures which can be factored out from domain-specific variations. We term this set of middleware functionalities "the middleware core" and use it as the basis of just-in-time customization.

2. We limit our discussion to the remote-procedure-call based middleware. The support of the full spectrum of middleware types and properties through *JiM* is an interesting and future research endeavor.

3. The *JiM* paradigm is most effective when the middleware is parasitical, i.e., part of the user application stack that responsible for managing distributed concerns. The paradigm can also be used in providing specialized active distributed services tailored for specific applications or domains. We defer to future research the discussion of the properties of *JiM* in the case of generic active distributed services. To these services, the distribution needs of individual user applications generally are not sufficient in deciding the included functionalities of middleware.

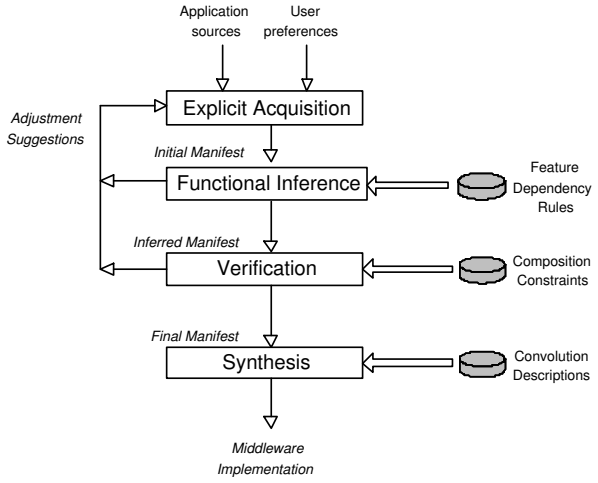

Figure 1: *JiM* customization stages

3.2 Overview

Just-in-time customization consists of four high-level stages as illustrated in Figure 1. The process is bootstrapped by the explicit acquisition of middleware functionalities through either the analysis of the application source or explicit preference indication by the user. A tool should be provided to facilitate this process through the use of mappings between configuration options and elements of user programs. The initial manifest of middleware features gathered from this stage is then further analyzed in the *Functional Inference* stage. The inference algorithms autonomously adjust the manifest in compliance with feature dependency rules. The "inferred manifest" is then validated according to external constraints in the *Verification* stage. The violations of either dependency rules or constraints re-initiate the *Explicit Acquisition* stage where users can adjust the desired feature set according to the violation information. The finalized manifest is passed to the *Synthesis* stage where the tailored middleware is synthesized according to *convolution descriptions* and a well-defined *coding naming schema*. These descriptions and the schema enable the selection of the right components of a particular feature implementation if this feature interacts with unselected features.

3.3 Dependencies, constraints, and convolution descriptions

In *JiM*, three types of specifications are defined in addition to the middleware implementation: feature dependencies, composition constraints, and convolution descriptions. A *feature dependency rule* states that the implementation of certain functionality is composed from other functionalities. These dependencies must be satisfied by adding the constituent functionalities if a composite functionality is included in the manifest. For example, selecting the mid-

dleware data type `String` would result in both `String` and `char` being included in the feature manifest if the implementation of `String` is based on that of the `char`. Dependency rules can be either automatically derived from feature implementations or manually specified to express logical relationships among features. *Composition constraints* are also dependencies that dictate the inclusions and exclusions of functionalities reflecting certain external conditions. Different from functional dependencies which are propositional, constraints are predicates on conditions independent of middleware itself. For example, targeting a cellphone application on J2ME[6] platforms requires the omission of certain functionalities due to the lack of support in J2ME virtual machines. This omission can be specified as a constraint to validate a manifest generated for cellphone applications. *Convolution descriptions* are also propositions describing the interactions among features. Convolutional descriptions are different from functional dependencies as the latter pertains to the implementations of functionalities and the former to their interactions. Convolutional descriptions are used at the synthesis stage to guarantee that the right interaction code is selected. This mechanism is explained in detail in Section 3.5.

3.4 Functionality acquisition

The goal of functional acquisition is to obtain a tailored and minimum set of middleware functionalities as desired by the user application. The process is divided into two stages: direct collection of functionalities (explicit acquisition) and the adjustment according to functional dependencies (inference).

3.4.1 Explicit acquisition

There are two basic forms of explicit acquisition: *automatic feature extraction* and *user preference indication*. Automatic feature extraction is to examine the program source and to detect middleware functionalities user applications intend to use. This intent can be discovered most explicitly in service declaration descriptions expressed in well-defined languages such as the interface definition language (IDL) in the CORBA and DCOM world, remote interfaces of the Java RMI mechanism, as well as the web service description language (WSDL) of Web Services. The language elements of IDL, for instance, embody a rich set of middleware functionalities including data types, synchrony/asynchrony of message passing, invocations styles, and many others. Remote service descriptions can serve as the initial recipe for determining the final ingredients of the synthesized middleware. In many cases, service descriptions alone are not sufficient to determine all the necessary functionalities because some are present, not as language elements, but as libraries or extension points for controlling the behaviour of middleware itself. One example is the interceptor infrastructure in CORBA implementations. The use of these types of functionalities can only be detected by analyzing the application source code. Another type of explicit acquisition is to allow the user to select additional features in foreseeing their future uses. A tool can be provided to facilitate both compiler-based and user-driven acquisition.

3.4.2 Rule-based inference

The functionality manifest established through the explicit acquisition typically will not be the final manifest used in synthesis. An inference process is needed to reason about functional dependency rules and to possibly include additional functionalities. This process must provide two guarantees regarding the inferred manifest:

1. Each functionality in the manifest must have its functionality dependency satisfied.

2. The inferred manifest must be a minimum superset of the explicit manifest.

The second garantee is critical to the *JiM* paradigm and also difficult to implement because the term "minimum" can be interpreted differently in different contexts. An appropriate cost model can be employed by inferrence algorithms to rank all possible supersets. The cost functions in the model can reflect the weights of many properties regarding a particular functionality such as its physical size, memory requirements, energy consumptions, level of preference, and others. The definition of the cost function can either be user-defined or provided by the middleware vendor. We will describe such an algorithm and the associated cost function in Section 4.2.

3.4.3 Verification

During the verification stage, the inferred manifest is checked against composition constraints. Though logically separate from the functionality acquisition mechanism, verification is frequently invoked in the inference stage to validate the inferred manifests. Violations of composition constraints can either be used to prune choices of feature supersets or to reinitiate the acquisition process with the information about offending functionalities and violated constraints. This requires different indications of user preferences or modifications of the application in the worst case.

3.5 Target synthesis

During the target synthesis stage, the tailored middleware is composed in strict accordance with the final manifest generated at the inference stage. Due to the convolution phenomena, it is often not correct to naively select the entire functionality. Instead, we need to select its relevant components in the context of a specific final manifest. This partial selection could be different from synthesis instance to another. To tackle this difficulty, we propose to use a structural schema to organize the code space of middleware features. The convolution descriptions guide the feature selection process. We discuss these concepts abstractly in the following sections and provide our specific implementations in Section 4.2.

3.5.1 Structural schema

The primary purpose of the structural schema is to make the code space of middleware features comprehensive to the synthesizer. The traditional structure of the code space is based on name spaces (or packages in Java terms) and, therefore, suited for the vertical dimension. Features modularized as aspects are non-hierarchical and convolutional. Therefore, the structural schema for *JiM* must make the following properties explicit for each aspect: the differentiation between *implementation* and *interaction*, and the set of *binary relationships* with other aspects. The former is necessary because, if the feature is selected, the implementation of aspects is included completely, but the interaction logic

[6]Java Micro Edition. URL: `http://java.sun.com/j2me`

might subject to partial selection. The explicit binary relationships in the schema allow the synthesizer to select the correct parts of an aspect according to what other aspects it interacts with as specified in the convolution descriptions. Instantiations of the schema elements, i.e., *implementation*, *interaction*, and *binary relationship*, can be either source files or bytecode representations[7].

3.5.2 Implementation selection

The implementation selection is the primary role of the *JiM synthesizer* which, according to the final manifest, determines and includes the correct constituents of a feature implementation in the final middleware implementation. The output of the synthesizer can either be a build configuration or an executable middleware instance. Utilizing the structural schema and the convolution descriptions, the mechanism of the synthesizer can be straightforward: if a feature is present in the manifest, the synthesizer firstly includes its implementation then its individual binary relationship with other features if they are also activated, by walking the code space according to the schema.

3.6 Functional verification

The functional correctness of middleware is typically verified through integration tests in which individual tests are provided for all the packaged functionalities. In the *JiM* approach, such integration tests are difficult to devise since the functional ingredients of the final middleware implementation change from case to case. However, we can take advantage of the high degree of orthogonality among the semantics of horizontal features and design test cases only for these individual features. It is then the synthesizer's job to automatically assemble the composite test plan from individual tests for the final middleware implementation. Functional verification is a separate logical step in completing the *JiM* process. It can be integrated as part of the synthesizer if we incorporate the source code of test cases into the structural schema described previously. We present such a testing framework based on JUNIT [8] and AspectJ in Section 4.2

4. JUST-IN-TIME MIDDLEWARE: THE ABACUS IMPLEMENTATION

In this section, we present the Abacus, the aspect-oriented ORBacus, as the prototype implementation of *JiM* to validate key *JiM* concepts that serve as the basis of our evaluation. Abacus is based on our long term aspect oriented refactoring of ORBacus, an open source industrial strength CORBA implementation. We have re-structured more than 65% of the original implementation and re-modularized them as aspects. In addition, we have substantially extended our aspect-aware IDL compiler [15] and built Arachne, the Abacus synthesizer. The following sections first give an overview of the characteristics of the Abacus architecture. The implementation of *JiM* concepts is then discussed in detail.

4.1 Architectural highlights

The architecture of Abacus is based on a very high level of configurability along two dimensions: the vertical dimension by preserving the original hierarchical configurability, and the horizontal dimension by modularizing crosscutting concerns. This high degree of configurability is the foundation of Just-in-time customization.

4.1.1 Enhancements of vertical configurability

The vertical configurability, i.e., changing parts of the hierarchical structure, is well designed in the original ORBacus implementation and also entirely preserved. Abacus can be vertically configured through two traditional ways: system property tables, which is the Java equivalence of the environment variable mechanism; and policies, which is a CORBA specific way of fine tuning the functionalities of the ORB[9]. We enhanced this configurability by adding support for J2ME environments through the creation of the J2ME network transport level support according to the MIDP2.0 specification[10]. This transport can be selected through the system property table.

4.1.2 Fine-granular horizontal configurability

A vast amount of aspects are available for configuration including IDL language data types, synchrony of remote invocation semantics, CORBA infrastructures, and programming styles. Table 1 categorizes a total of 26 aspects modularized in Abacus.

Categories	*Features*
Server Data Types	Integer, String, Double, Fixed, Float, Long Wide Character, Wide String
Remote Invocation Semantics	Synchrony, Asynchrony (Oneway), Passing-by-value
CORBA Infrastructures	Policies, Implementation Repository, Interface Repository, Type Code, Collocation Optimization,Interceptors, Caching, Fault Tolerance, Object Disposal
Remote Invocation Styles	DII, DSI, Any, Dynamic Any
Regional Support	Locale, Encoding conversion

Table 1: Complete Listings of Configurable Aspects

4.2 Arachne: The Abacus synthesizer

The various stages of Just-in-time customization in Abacus are integrated in and largely automated by a tool we have built named Arachne[11]. Arachne is composed of three components: the aspect-aware IDL compiler, the Java source parser, and the inference engine. The aspect-aware IDL compiler produces more modular skeletons and stubs in both classes and aspects. It also performs the initial feature selections by examining the language elements used in the IDL declarations. The Java source parser further selects features through checking Java class types used in the user application. Currently, the just-in-time vertical configuration is

[7]We think it can also be linkable executables. However, the current AOP compilers only support weaving either source code or byte code.

[8]JUnit. URL: `http://www.junit.org`

[9]For the rest of the paper, we use ORB (Object Request Broker) to denote the implementation instance of CORBA

[10]J2ME Mobile Information Device Profile URL: `http://java.sun.com/products/midp/`

[11]A mythical character who is too good at weaving

not yet taken into consideration since configurations via environment variables, policies, or even deployment descriptors typically happen post development time. We defer the discussion of the "later-time" just-in-time customizations of both dimensions to our future research.

4.2.1 Aspect-aware IDL compiler

The aspect-aware IDL compiler carries out two key functions in the early stage of the *JiM* process: aspectizing stubs and skeletons and explicitly acquiring middleware functionalities from service declarations. Stubs and skeletons are integral parts of the middleware, generated by IDL compilers as adapters that translate application semantics into the machinery of middleware. Therefore, aspect-oriented modularization of stubs and skeletons must be performed during IDL compilation by properly generating the minimum (convolution-free) adaptation code and expressing the rest of the functionalities in aspect modules. During the same process, the compiler conveniently collects all the IDL language elements that can be used to initiate functionality inference. Our aspect-aware IDL compiler is an extension of a prototype presented in [15] and constructed entirely in aspects woven into the JacORB [12] IDL compiler. Our compiler is still experimental since we do not have access to the source of the ORBacus IDL compiler.

4.2.2 Dependencies, constraints and convolution descriptions

In Arachne, dependencies, constraints and convolution descriptions are separate specifications defined as Boolean expressions. Dependencies and convolution descriptions are propositional reflecting the innate relationships among middleware aspects. Constraints are predicative because the prescribed relationships are only true under specific external conditions. Constraints can be either provided as references by the vendor or defined by the user.

Currently, dependencies in Arachne are specified in a simple language called horizontal dependency definition language (HDL). Figure 2 illustrates the basic structure of HDL. Identifiers of aspects can be declared using the `aspect` keyword (line 1). Dependency rules of the same nature can be grouped and scoped with a name (line 2). "*" stands for logical `AND`, and "+" stands for logical `OR`. In our example (line 3), `aspect1` depends on `aspect2` together with either `aspect3` or `aspect 4`. In Figure 3, we list all the dependencies of horizontal features in Abacus. An exception is made for the dependency rule "core" (first rule in Figure 3) in which "core" is not a selectable feature and always activated by default. The "core" dependency rule allows us to specify the minimum valid configuration of Abacus. In our current definition, a valid Abacus instance must support either synchronous or asynchronous communication (oneway). Constraints are used to limit the freedom of feature selection by denying the selection of certain features in certain environments. They are also defined in HDL format in which the left side of the Boolean expression is the name of the constraint rather than the name of a feature. In Figure 4, we list constraints for both J2ME and desktop platforms defined in two HDL modules. The J2ME constraints are mostly reflections of platform incompatibilities between J2SE and J2ME. The division into separate constraining rules is to provide more specific feedbacks to users upon violations. The Desktop constraints are illustrative. Table 2 gives the descriptions of the constraints.

[12] JacORB URL:`http://www.jacorb.org`

```
1 aspect aspect1, aspect2, ... ;
2 modulename {
3       aspect1 = aspect2 * (aspect3 + aspect4)
4       .....;
5 };
```

Figure 2: Illustration of language elements in HDL

```
aspect any, dynamic, pi, synchrony,
asynchrony, ...;
Core {
  core = synchrony + asynchrony;
  any = int + string + ulonglong + ... ;
  dii = dynamic;
  dsi = dynamic;
  pi = dynamic;
  dynamic = any;
  ulonglong = longlong ;
  IR = (DII + DSI)*typecode;
  convert = locale;
  wstring = wchar;
};
```

Figure 3: Dependency specifications in Abacus using HDL

Constraint	*Annotations*
Resource	J2ME disallows object serialization used by "valuetype" and user garbage collection
Precision	J2ME does not support high precision types such as "double" and "fixed"
Infrastructure	The applet infrastructure is not supported in J2ME
Reflection & type info	Reflective remote invocations, often assisted by type information are commonly used in enterprise applications

Table 2: Descriptions of Constraints

Convolution descriptions describe the one-to-many relationships between any two horizontal features. Figure 5 gives a complete listing of the convolution descriptions used in Arachne. The left side of the Boolean expression is a selectable feature, which crosscuts the features on the right side. These descriptions are essential to the final synthesis as they direct which part of the "weaving" code of a particular feature should be selected with respect to the activated features it crosscuts.

4.2.3 Inference algorithm

Given the initial feature manifest, the inference algorithm in Abacus is responsible for selecting unspecified features in order to satisfy all functional dependency rules. If multiple choices for selections exist, the algorithm guarantees to generate a minimum superset of the initial manifest.

Before we introduce the algorithm, we first define a few terms. *Feature state*: A feature can be in one of three states during the inference process: selected (`true`), ex-

```
j2me {
      resource = ! valuetype * ! finalize;
      precision = ! double * ! fixed;
      infrastructure = ! applet;
};

desktop {
      reflection = ! dii  * ! dsi ;
      typeinfo = ! IR + ! IMR ;
};
```

Figure 4: Constraint specifications in Abacus

```
convolution_descriptions {
  double = any * valuetype;
  finalize = DII;
  fixed = any * IR;
  IR = policy * PI;
  policy = IMR * oneway * PI;
  ulonglong = any * valuetype;
  any = IR * policy * PI * valuetype;
  dii = any;
  dynamic = IR;
  longlong = double * any * valuetype;
  oneway = collocation;
  PI = DII * DSI * IR;
  wchar = any * convert * valuetype;
};
```

Figure 5: Convolution descriptions in Abacus

cluded (`false`), and unspecified (`unknown`). *Rule activation*: A dependency rule r is *activated* if the feature on the left side of r is selected, i.e., its value is "1" in the manifest. *Minterm*: Our algorithm only deals with rules written in the form of sum-of-products. Each product term is called a *minterm*. A minterm represents an `AND` relationship between n features in the term. *Cost function*: A cost function associates each feature with a positive number representing the cost of including this feature. In our evaluation, we associate the cost with the bytecode size of each feature to determine the preferred alternative configuration. The cost could be other metrics of resource consumptions or even selected by users.

We first present a simple algorithm, Algorithm 1 requires all rules having only one minterm evaluated to `unknown`. The algorithm simply iterates each dependency rule r and invokes the `evaluate` function on r against the inferred manifest. The function returns either `true` (rule satisfied) or `false` (rule violated) or `unknown`. In the case of `unknown`, new features are added to the manifest to satisfy r. The algorithm stops if no new features are added after iterating through all rules in consecutive runs. The complete algorithm handles multiple disjunctive minterms in dependency rules by recursively invoking the simple algorithm on each minterm. At each recursive step, the cost function is used to determine the minimum configuration and pass it to the upper recursive level. Due to the page limitation, we present the full description of the algorithm and the proof for its guarantees in an extended version of this paper.

4.2.4 Packaging schema

Due to the convolution problem, the code of an aspect needs to be packaged according to a well-defined schema so that the relevant parts of the aspect can be correctly

Algorithm 1 Simple Inference Algorithm

Require: initial manifest F, dependency rules R
  Let $F' = F$
  loop
    for each $r \in R$ **do**
      if r is activated by F' **then**
        `Boolean` result = evaluate(r, F')
        if result = `true` **then**
          GOTO next r
        else if result = `false` **then**
          RETURN error
        else
          for each unkown f in r **do**
            set f to 1 or 0 so that r evaluates to 1
          end for
        end if
      end if
    end for
    if $F' = F$ **then**
      BREAK
    else
      $F = F'$
    end if
  end loop
  Set all unknown features in F to `false`
  RETURN F

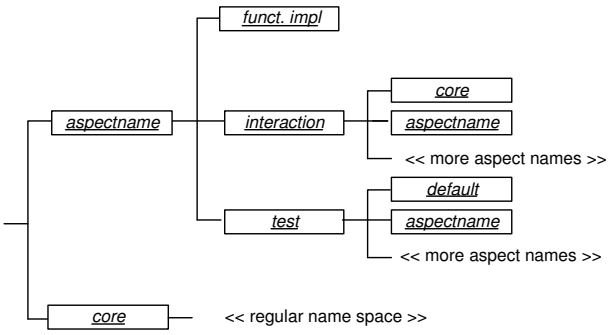

Figure 6: The packaging schema in Abacus

selected for the final synthesis. Figure 6 illustrates the general definition of the schema. In this hierarchical schema, the implementation and the interaction of an aspect, represented by `aspectname`, are separated into two sub-trees, `functional implementation` and `interaction`. The specific interactions between any two aspects is organized using one aspect's identifier as the package name under the "interaction" sub-tree of the other aspect's root. For simplicity, we use a Java package hierarchy to instantiate this schema.

4.2.5 Synthesis process

The synthesis process is guided by the Arachne user interface and initiated by loading IDL definitions and application source locations. During the source analysis, we link the class types in the application source to the implementation class types of an aspect to discover whether this aspect should be activated or not. Collecting type information of Java applications can be accomplished by many tools, and, for convenience, we chose to use the parsing engine

of Prism [16], our aspect mining tool. Upon the completion of IDL and source analysis, the activated features are marked on the feature pallet on which user can make further selections and adjustments. The complete set of explicitly acquired features is then processed by the inference engine which generates the final manifest of features.

With the help of our packaging schema and convolution descriptions, the final synthesis procedure proceeds as described in Algorithm 2. The output of this procedure is a list of modules corresponding to the final manifest. In our current implementation, a resource list is generated and handed over to the AspectJ weaver. More sophisticated building tools such as ant[13] could also be exploited. A note-worthy characteristic of our HDL-based feature synthesis procedure is that it is easy to add a new aspect feature if its implementation conforms to the code naming schema. Arachne is able to reason about this new feature if its dependencies and convolution descriptions are correctly specified in HDL.

Algorithm 2 Synthesis Procedure

Require: final manifest F, convolution descrition set C
  for all f in F **do**
    if $\exists c \in C$ such that f activates c **then**
      for each f' on the right side of c **do**
        Include all modules in the path "$f \Rightarrow functionalimplementation$
        Include all modules in the path "$f \Rightarrow core$
        Include all modules in the path "$f \Rightarrow interaction \Rightarrow f'$"
      end for
    end if
  end for

4.2.6 Integration test synthesis

The functionality of the synthesized Arachne instance can be verified through a synthesized set of test cases for the features included in the implementation. In Abacus, test cases are written in JUnit independently for each feature. All share a common starting point by registering themselves with this starting point through AspectJ advices. The schema for test-case organization is the same as for the features, and the same synthesis procedure (Algorithm 2) is used. Therefore, when a particular feature is selected, its test cases are also "woven" into the final test plan.

5. EVALUATION

Our evaluation focuses on assessing the customization capability of *JiM* concepts through the Abacus implementation. This assessment includes a broad range of software metrics including both static characteristics and dynamic performance evaluations. We divide the evaluation to two portions: first the exemplary usage of Abacus in supporting a ubiquitous messenger application and then the randomized selections of features which models different real-world usage scenarios. All experiments are run on a Pentium 4 2.5GHz Linux workstation running Redhat 9.0. The J2ME configuration is evaluated using the Nokia Series 90 concept emulator[14].

[13]Ant. URL: `http://ant.apache.org/`
[14]Nokia Emulators. URL: `http://www.forum.nokia.com`

```
1  #include<mmsg.idl>
2  module CORBAMessenger{
3     interface MiniMessenger{
4          void login(in string user);
5          void logout(in string user);
6          seqMsg contact(in Msg info);
7          oneway void asyncContact(in Msg info);
       };
};
```

Figure 7: Service declarations of UMessenger

5.1 Ubiquitous Messenger

A distributed application is well-suited for *JiM* if the same operational logic ought to be supported in diversified domains. Instant messaging applications such as ICQ or the MSN messenger are good examples because their primary functionality, i.e., message exchange, is widely available on networked computers including cell phones, PDAs, private desktops and enterprise environments. Each environment has distinct computational requirements. We have designed a messenger application, ubiquitous messenger (UMessenger), which provides increasingly richer functionalities in environments with lesser resource constraints. Table 3 shows the setup of our target environments, the assumed constraints for our messenger applications and the aspect configurations.

Target Environment	*Constraints*	*Aspect Configurations*
Embedded Platform (J2ME, CLDC 1.0, MIDP 2.0)	Only supports strings and numeric data	sync, async, string,long long
Desktop (J2SE)	Supports large file exchange	All of above + double, valuetype
Enterprise Features	Supports dynamic types, encryption, and transaction through interceptors	All of above + Any, portable interceptors

Table 3: Ubiquitous Messenger Setup

Figure 7 shows the service declarations in IDL definitions for the messenger in these three target environments. The remote interfaces are kept simple and identical in all cases, but we use richer sets of IDL data types in the definitions of the message payload to reflect different functionalities of the messenger. Figure 8 shows the corresponding IDL definitions of messages. Despite its simplicity, we use UMessenger to show the kind of versatility in aspect configurations Abacus is able to support.

5.1.1 Static properties

We first measure the static properties on the compiled Java byte code. The static properties such as size and coupling are generally considered as reflecting the complexity of the code and the degree of difficulties in terms of maintenance and evolution. Table 4 reports the sizes of the synthesized ORBs as well as the coupling metrics measured on bytecode by JDepend[15]. Our baseline of the compari-

[15]JDepend URL:`http://www.clarkware.com/software/JDepend.html`

```
module CORBAMessenger{
      struct Msg{
         string from;
         string to;
         string stringdata;
         long long numericData;
      };
};
```

```
module CORBAMessenger{
       valuetype LargeData{
          private string filename;
          typedef sequence <octet> content;
          long save();
       };
       struct Msg{
         //repeat fields of minimum message
         sequence <double> floatData;
         LargeData payload;
       };
};
```

```
module CORBAMessenger{
       valuetype LargeData{
        //same as in desktop message
        };
       struct Msg{
       //repeat fields of desktop message
        any anyData;
       };
};
```

Figure 8: Message definition for embedded environment (a), desktop environment (b), enterprise environment (c)

son is the original ORBacus ORB which does not adapt to different functionalities in UMessenger. As expected, our synthesized ORB constantly has smaller bytecode sizes. For example, the configuration for J2ME produces an ORB measured 42% of its original size. The degree of coupling in all three configurations is considerably lower than in the original ORB varying from 16% to 33%.

Target Environment	*Bytecode Size*	*Efferent Coupling*	*Afferent Coupling*
Embedded Platform	823.9k	249	183
Desktop (J2SE)	875.1k	251	185
Enterprise	1168k	304	230
Original	1945.3k	365	275

Table 4: Static comparisons with the original ORB

5.1.2 Runtime properties

To study the runtime characteristics of Abacus, we first measure and compare the time taken for payloads to traverse the middleware stack for all three generated ORBs as well as the original ORB. We then collect the response time, the memory usage, and the cache miss rates of these ORBs.

Payload transportation. For this experiment, we host both the client and the server on the same machine to minimize the network delay. Figure 9 shows the average cost, measured using a regular JVM, of the payload transportation for the original ORB as well as the synthesized ORBs for the UMessenger in the desktop and the enterprise settings. The full J2ME configuration is measured in an emulation setting using the Nokia Series 90 emulator. The response

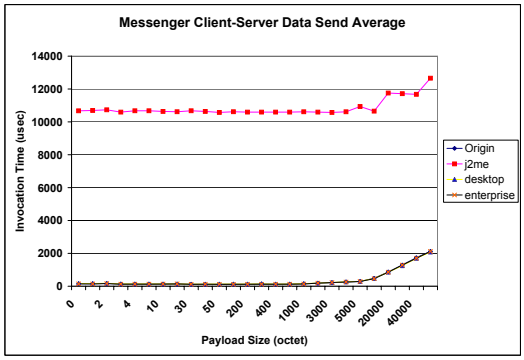

Figure 9: Data transport of UMessenger using Abacus

time of ORBs in the regular JVM is roughly equal as the graph shows. The synthesized ORBs incur virtually no overhead compared to the original version in the presence of the AspectJ runtime infrastructure. We are not observing apparent speedups. This is a combined effect of both the efficient design in the original implementation and the difficulty in performance gains in face of the JVM overhead and optimization. The emulated J2ME responses are significantly slower compared to regular JVM environments.

Response, resource, and cache. Table 5 reports a few runtime metrics we have collected in addition to the benchmark. We measure the average response time, memory usage and cache performance numbers, all compared to that of the original version. The cache misses are collected for the server process using PCL[16] over 200,000 remote invocations. The response time is measured as the average of 200k remote invocations of an empty method("ping"). All three configurations are basically equivalent in terms of the response time with the J2ME version using regular network transport being 7% faster than the original ORB. The memory usages of all three versions are within 1% difference of each other. However, compared to the original version, Abacus exhibits an average of 24% reduction of memory usage. Similar patterns are observed on Level-1 cache misses, including both data cache (DL1), instruction cache (IL1), and Level-2 miss rates. The J2ME configuration consistently has the best performance measures because it has a minimal set of functionality.

Target	*Response*	*Memory*	*IL1*	*DL1*	*L2*
J2ME	261	15223k	64067k	190394k	0.0058
Desktop	281	15164k	69582k	195186k	0.0052
Enterprise	277	15028k	75307k	197084k	0.0052
Original	282	19694k	70277k	210822k	0.0063

Table 5: Dynamic Properties Comparing with Original ORB (Response time in μ sec)

[16]Performance Counter Library. URL:`http://www.fz-juelich.de/zam/PCL/`

5.2 Randomized Customization

The purpose of UMessenger is to illustrate one type of application which can take advantage of the customization capabilities of *JiM*. Our intention of evaluating Abacus on different application domains has influenced the fabrication of features in UMessenger. And this is not an approximation of the customization choices that the real world demands of *JiM*. To emulate unpredictability of user application features, we choose to randomize the selections of features by assigning each configurable feature a random variable from a uniform distribution. Each feature has equal chances of being selected, unselected, or unknown. We iterate this randomization for fifteen times and obtain ten valid configurations. Table 6 shows the initial random selections (under R header) and the selection decisions made by our inference engine (under I header) in five random runs. A selected feature is represented as "1", or "0" otherwise. "x" means the selection is unspecified. In the "I" column, we only show the value if it is modified compared to its initial value in the "R" column. We can observe a few characteristics of Arachne: 1. the selection decisions of all features are made; 2. most of the "x" features are set to "0" driven by the minimalism goal of our inference algorithm.

Features	Random Configurations									
	RC1		RC2		RC3		RC4		RC5	
	R	I	R	I	R	I	R	I	R	I
double	x	0	x	0	1		x	0	1	
finalize	x	0	x	0	0		0		0	
fixed	x	0	x	0	1		1		x	
IMR	x	0	x	0	1		0		1	
IR	0		0		0		0		x	
policy	x	0	0		x	0	0		0	
ulonglong	0		x	0	1		x	0	x	
any	1		x	1	0		1		x	
applet	0		1		1		1		x	
collocation	x	0	x	0	x	0	1		0	
convert	0		1		x	0	x	0	x	
DII	x	0	x	0	x	0	x	0	1	
DSI	1		0		0		1		x	
oneway	1		x	0	1		1		0	
PI	1		1		0		x	0	1	
valuetype	x	0	x	0	x	0	x	0	x	
wchar	1		x	0	1		1		x	
dynamic	x	1	x	1	x	0	1		0	
float	x	0	1		x	0	0		x	
longlong	0	0	1		1		x	0	x	
Validity	valid		valid		valid		valid		invalid	

Table 6: Randomized Feature Selection and Synthesis in Abacus

5.2.1 Static properties

We collect the same static properties as in the previous case. We show in table 7 that the bytecode sizes of six randomly synthesized ORBs are between 56% (RC3) and 75% (RC4) of the original ORB, with the efferent coupling ranging from 73.6% (RC3) to 88% and the afferent coupling ranging 72.6% to 88.7%. This shows the redundancy in the original implementations and Abacus can always synthesize a leaner middleware implementation.

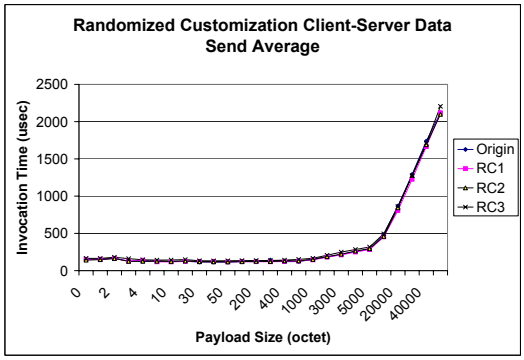

Figure 10: Benchmarking the Randomly Synthesized ORBs

Target	*Bytecode Size*	*Efferent Coupling*	*Afferent Coupling*
RC1	1237.8k	299	225
RC2	1217.1k	300	224
RC3	1095.6k	269	197
RC4	1463k	322	244
RC5	1290k	309	233
RC6	1351k	313	236
Original	1945.3k	365	275

Table 7: Static Comparisons with Original ORB

5.2.2 Runtime properties

To study the runtime properties of the randomly synthesized ORBs, we repeat the same set of experiments by first measuring the average time for transporting payloads of increasing sizes. We then measure the response time, the memory consuption, and the cache miss rates. All experiments are run in the regular JVM.

Payload transportation. In Figure 10, we plot the average time of the payload transportation for randomly synthesized ORBs. Similar to the case of UMessenger, the performances of all versions are roughly equal. This further illustrates the ORBs synthesized in AspectJ incur no apparent overhead.

Response, resource, and cache. In Table 8, we show the average response time, memory usage, and cache misses for the six randomly synthesized configurations. Similar to the case of UMessenger, the response time of all configurations are within 1% differences of each other. The memory usages reductions are within 5% (RC4) and 24% (RC3) as compared to the original version. The reductions of level-1 instruction cache misses is as much as 8% (RC6) and 3% on average. The reductions of level-1 data cache misses range from 1% (RC3) to 10% (RC6). The average reduction of level-2 cache-miss rates is 4%. The large majority of the

synthesized ORBs perform better than the original one.

Target	*Response*	*Memory*	*IL1*	*DL1*	*L2*
RC1	285	15473k	67518k	194102k	0.0064
RC2	287	15175k	69210k	190411k	0.0055
RC3	284	14960k	68730k	208584k	0.0056
RC4	268	18660k	70895k	205221k	0.0066
RC5	275	15802k	67391k	190998k	0.0062
RC6	281	15231k	64723k	189562k	0.0060
Original	282	19694k	70277k	210821k	0.0063

Table 8: Runtime Properties Comparing with Original ORB

5.2.3 Concluding remarks

Our evaluation shows that the *JiM* paradigm is effective in solving this design dilemma: achieving generality while sacrificing specialty or vice versa. Abacus is general enough to fit dramatically different distribution concerns and yet sufficiently specialized for individual scenarios. Both static and dynamic measurements reflect that we can have significant complexity reductions and runtime improvements if we tailor middleware according to specific distribution needs of individual applications.

6. RELATED WORK

Configurability and customization are active themes of current middleware research. We skip the comparisons to research work in [10, 4, 2] because they address *adaptation*, which can be interpreted as dynamic customization. Since adaptation is not the focus of this paper, we first discuss the related research work in the context of highly customizable middleware architectures. In addition, we discuss differences between the synthetic approach of *JiM* and the product line approaches.

Customizable Middleware: The FACET [8] project experiments with the concept of feature subsetting in middleware through the aspect oriented implementation of a CORBA event channel. FACET divides the functionality of the even channel into the "base" and "features" where the "base" implements the fundamental functionality of FACET and "features" are added into the "base" via AspectJ. A large number of configuration possibilities exist in FACET, and feature dependency graphs are used to limit valid configurations. Fundamentally different from FACET, Abacus is proposed as a post-postulated solution where the distribution concerns of applications are evaluated and reasoned. In terms of implementation techniques, we share some common design elements, such as the core-based design, the use of feature dependencies, and the aspect oriented test case synthesis. In Abacus, through dependency specifications, we make very explicit both the relationships among core and features as well as the relationships among features. This serves the foundation for automatic inference and synthesis.

The MicroQoSCorba project [12][17] targets resource-constrained environments and provides both a fine-grained architecture and CASE tools to make automatic adjustments according to the variations in both user applications and platforms. Like Abacus, MicroQoSCorba also uses IDL compilers to collect required middleware features and to direct the building process to select relevant implementations. However, the source code analysis is missing in MicroQosCorba compared to Abacus. The feature dependencies and the inference stage are not explicitly addressed in MicroQosCorba. Most importantly, Abacus provides a higher degree of modularity using AOP and scales in two dimensions. We have similar approaches to MicroQoSCorba of making key middleware elements lightweight. However, contrary to MicroQoSCorba, our "customization" is not at the cost of losing the original full functionality, benefiting from the aspect oriented approach. As we have illustrated, the configuration domains of Abacus are not limited to embedded systems and also include other application domains.

Product Lines and Generative Programming: GenVoca [1] advocates the synthesis of a family of complex systems from incrementally adding features to simple ones using step-wise-refinement (SWR). The AHEAD tool suite allows algebraic specifications of the SWR relationships among features and carries out the automatic synthesis. GenVoca embodies a rich yet comprehensive methodology for programming specification and program generation. Comparatively, our research has two slightly different focal points: we focus on enabling the user-driven middleware composition from fine-granularity modules, a paradigm orthogonal and complimentary to the synthesis-based approaches in GenVoca and SWR; we focus on the higher degree modularization in large and complex software systems such as middleware through the effective use of aspects.

Colyer and Clement [5] demonstrated how aspect oriented refactoring can be used to support commercial middleware product families. The focus of their work is to evaluate the suitability of AOP in supporting very large software projects through capturing crosscutting features in aspects and weaving them into other product family members as new features. They provide valuable experiences and insights which benefit us in scaling *JiM* to support even larger size middleware implementations.

7. CONCLUSION

Traditionally, the middleware architecture and its hosted applications are bound by a causal relationship: middleware is designed, implemented and packaged first; applications are developed in accordance with services provided by a particular middleware implementation. This type of "*pre-postulated*" middleware always poses many challenges in customizing middleware functionalities due to the difficulties in making good presumptions of what features are actually needed in specific usage scenarios. We believe that, in the case of middleware customization, the best presumption is not to make any presumptions at all. We have proposed to reverse the traditional causal relationship by delaying the composition of middleware architecture till after the user application is designed or implemented. We term middleware architectures supporting this new relationship Just-in-time middleware architectures (*JiM*).

We have presented the mechanism of *JiM* as a multi-staged process including feature acquisition, functional inference, constraint verification, and implementation synthesis. We also have built Abacus, a Java ORB based on "aspectizing" ORBacus, an industrial strength CORBA implementation. In addition, we have built Arachne, a tool which

[17] MicroQoSCORBA. URL:`http://microqoscorba.eecs.wsu.edu`

integrates the aspect-aware IDL compiler and the inference engine. Our evaluation proves that Abacus is capable of solving the "impedance mismatch" problem by providing middleware functionalities across application domains from embedded devices to enterprise environments through post-compilation customization. Our quantifications show that Abacus exhibits a great versatility in terms of physical sizes, resource consumptions, and runtime performance, in addition to general improvements on these qualities as compared to the original implementation.

Our title of this paper suggests that we are just at the beginning of exploiting this useful paradigm. Our current implementation has greatly benefited from but is also limited by the design of ORBacus. That is, our customization ability is confined within the capability of the original ORBacus. In our future research, we will focus on further validation of the *JiM* approach through supporting more diversified middleware features and more application domains using both new implementations and case studies. We are also interested in exploiting the load-time weaving capability of AspectJ and postponing the synthesis of middleware implementations in order to take advantage of the load time information. For instance, we can define cost functions taking memory usage as inputs. Finally, we will also study how generative approaches can benefit *JiM* by incorporating model driven techniques such as MDA[18].

Acknowledgments

This research has been supported in part by an NSERC grant and in part by an IBM CAS fellowship for the first author. The authors are very grateful for this support.

8. REFERENCES

[1] D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004.

[2] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski. The Design and Implementation of Open ORB 2. *IEEE Distributed Systems Online Journal 2(6)*, 2001.

[3] Jr. Brooks, F.P. *The Mythical Man-month*. Addison-Wesley Publishing Company, 1975.

[4] M. Clarke, G. Blair, G. Coulson G., and N. Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'2001)*, November 2001.

[5] Adrian Colyer and Andrew Clement. Large-scale AOSD for middleware. In *3rd International Conference on Aspect-oriented Software Development (AOSD'04)*, pages 56 – 65, Lancaster, UK, 2004.

[6] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[7] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428. ACM Press, 1993.

[8] Frank Hunleth and Ron Cytron. Footprint and Feature Management using Aspect-Oriented Programming Techniques. In *Languages, Compilers, and Tools for Embedded Systems (LCTES'02)*, 2002.

[18]Model Driven Architecture. URL:`www.omg.org/mda`

[9] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[10] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.

[11] Thomas Ledoux. OpenCorba: a Reflective Open Broker. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*, pages 197–214, Saint-Malo, France, July 1999. Springer-Verlag.

[12] A. David McKinnon, Kevin E. Dorow, Tarana R. Damania, Olav Haugan, Wesley E. Lawrence, David E. Bakken, and John C. Shovic. A Configurable Middleware Framework for Small Embedded Systems that Supports Multiple Quality of Service Properties. Submitted to Software Practice and Experience.

[13] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using non-functional requirements: a process-oriented approach. *IEEE Transactions on Software Engineering*, 1992.

[14] Charles Zhang and Hans-Arno Jacobsen. Quantifying Aspects in Middleware Platforms. In *2nd International Conference on Aspect Oriented Systems and Design*, pages 130–139, Boston, MA, March 2003.

[15] Charles Zhang and Hans-Arno Jacobsen. Refactoring Middleware Systems: A Case Study. In *International Symposium on Distributed Objects and Applications (DOA 2003)*, Catania, Sicily (Italy), 2003. Lecture Notes in Computer Science, Springer Verlag.

[16] Charles Zhang and Hans-Arno Jacobsen. Prism is research in aspect mining. In *Companion of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2004.

[17] Charles Zhang and Hans-Arno Jacobsen. Resolving Feature Convolution in Middleware Systems. In *Proceedings of the 19th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, September 2004.