Model-driven Engineering of Component Systems

Krishnakumar Balasubramanian
kitty@dre.vanderbilt.edu

February 28, 2006

Contents

List of Figures

List of Tables

Abstract

While distributed object computing (DOC) middleware like CORBA and Java RMI were a significant improvement over prior middleware for developing distributed systems, there are significant limitations with DOC middleware. These include the inability to provide multiple alternate views of services on a per-client basis, inability to navigate between interfaces in a standardized fashion, low-level mechanisms for specification and enforcement of policies, complexity of middleware configuration and *ad hoc* deployment techniques. Standards-based component middleware like CORBA Component Model (CCM), Enterprise Java Bean (EJB) and Microsoft .NET improve upon the previous generation middleware by providing higher-level abstractions for expression and realization of design intent and flexibility of configuration. However, lack of system composition tools, complexity of the declarative platform notations and API, and composition overhead in large-scale component systems are significant limitations to the widespread adoption and usage of component technologies for enterprise distributed, real-time and embedded (DRE) systems.

This thesis proposal provides three contributions to the design and deployment of component-based enterprise DRE systems. First, it describes a domain-specific modeling language (DSML) toolchain that allows multi-level, flexible and scalable composition of systems. Second, it describes how the high-level abstraction provided by the DSML toolchain is used to automate generation of metadata for multiple component middleware platforms like CCM and .NET. Finally, it describes an optimization framework that is proposed to be built using the high-level abstraction of models. This optimization framework optimizes the performance and footprint of systems in an application transparent fashion, by exploiting the application context information available at the model level. To illustrate the platform-independence of the optimization framework, the optimizations will be prototyped, measured and validated in the context of more than one middleware platform, i.e., CCM as well as .NET Web Services.

Chapter 1

Introduction

1.1 Emerging Trends and Technologies

During the past two decades, advances in languages and platforms have raised the level of software abstractions available to developers. For example, developers today typically use expressive object-oriented languages such as C++ [1], Java [2], or C# [3], rather than FORTRAN or C. Object-oriented (OO) programming languages simplified software development by providing higher level abstractions and patterns. For example, OO languages provide support for associating data and related operations as well as decoupling interfaces from the implementations. Thus well-written OO programs exhibit recurring structures that promote abstraction, flexibility, modularity and elegance. Resting on the foundations of the OO languages, reusable class libraries and application framework platforms [4] were developed. This also led to the development of robust distributed object computing middleware (DOC) which applied design patterns [5] (like Broker) to abstract away low-level operating system and protocol-specific details of network programming. This resulted in the development of distributed systems since the DOC middleware hid a lot of the complexity associated with building such systems using previous generation middleware technologies. DOC middleware standards like CORBA [6] and Java RMI [7] coupled with mature implementations like TAO [8] led to development of more robust software and more powerful distributed systems. While DOC middleware provided a number of advantages over previous generation middleware, a number of significant limitations remain. Some of the limitations with DOC middleware include:

- **Inability to provide multiple alternate views per client.** An object in DOC middleware like CORBA typically implements a single class interface, which may be related by inheritance with other classes. In contrast, a component can implement many interfaces, which need not be related by inheritance. A single component can therefore appear to provide varying levels of functionality to its clients.

- **Inability of clients to navigate between interfaces of a server in a standardized fashion.** Components provide transparent "navigation" operations, *i.e.*, moving between the different functional views of a component's supported interfaces. Conversely, navigation in objects is limited to moving up or down an inheritance tree of objects via downcasting or "narrow" operations. It is also not possible to provide different views of the same object since all clients are granted the same level of access to the object's interfaces and state.

- **Extensibility of the middleware limited to language (Java, C++) and/or platform (COM, CORBA).** Objects are units of instantiation, and encapsulate types, contracts, and behavior [9] that model the physical entities of the problem domain in which they are used. They are typically implemented in a particular language and have some requirements on the layout that each inter-operating object must satisfy. In contrast, a component need not be represented as a class, be implemented in a particular language, or share binary compatibility with other components (though it may do so in practice). Components can therefore be viewed as providers of functionality that can replaced with equivalents components written in another language. This extensibility is facilitated via the Extension Interface design pattern [10], which defines a standard protocol for creating, composing, and evolving groups of interacting components.

- **Accidental complexities in configuation of middleware, specification and enforcement of policy.** Traditional DOC middleware provided very primitive mechanisms *i.e.*, low-level mechanisms for configuration of the middleware as well as specification of various policies. Since configuration and specification of policy was done using *imperative* techniques, it was typically done in the same language as that of the implementation. This led to the configuration of the middleware becoming complex, tedious and error-prone.

- ***Ad hoc* deployment mechanisms.** Deployment of systems using traditional DOC middleware is also done in an *ad hoc* fashion using custom scripts. The scripts were usually targetted at deploying a single system, and hence had to be rewritten for every new system, or in some cases for even different versions of the same system. The development and maintenance of this *ad hoc* infrastructure for deployment was an unnecessary burden on DRE system developers.

Thus it is clear that system developers have to face significant challenges when building complex enterprise DRE systems using DOC middleware. One promising solution to alleviate the complexities of traditional DOC middleware is component middleware technologies.

1.2 Component Middleware

Component middleware technologies like EJB [11], Microsoft .NET [12], and the CORBA Component Model (CCM) [13] raised the level of abstraction by providing higher-level entities like components and containers. Components encapsulate "business" logic, and interact with other components via *ports*. The different kinds of *ports* include:

- *Provided interfaces,* which are distinct named interfaces provided by the component. *Provided interfaces* enable a component to export a set of different functional roles to its clients.
- *Required interfaces,* which are interfaces used to specify relationships between components. *Required interfaces* allow a component to accept references to other components and invoke operations upon these references. They therefore enable a component to use the functionality provided by other components.
- *Event Sources and Sinks,*, which define a standard interface for the Publisher/Subscriber architectural pattern [14]. Event sources/sinks are named connection points that send/receive specified types of events to/from one or more interested consumers/suppliers. These types of ports also hide the details of establishing and configuring event channels [15] needed to support The Publisher/Subscriber architecture.
- *Attributes*, which are named values exposed via accessor and mutator operations. Attributes can be used to expose the properties of a component that are exposed to tools, such as application deployment wizards that interact with the component to extract these properties and guide decisions made during installation of these components, based on the values of these properties. Attributes typically maintain state about the component and can be modified by these external agents to trigger an action based on the value of the attributes.

Today's reusable class libraries and application framework platforms minimize the need to reinvent common and domain-specific middleware services, such as transactions, discovery, fault tolerance, event notification, security, and distributed resource management. For example, enterprise systems in many domains are increasingly developed using applications composed of distributed components running on feature-rich middleware frameworks. In component middleware, components are designed to provide reusable capabilities to a range of application domains, which are then composed into domain-specific assemblies for application (re)use. The transition to component middleware is gaining momentum in the realm of enterprise DRE systems because it helps address problems of inflexibility and reinvention of core capabilities associated with prior generations of monolithic, functionally-designed, and stove-piped legacy applications. Legacy applications were developed with the precise capabilities required for a specific set of requirements and operating conditions,

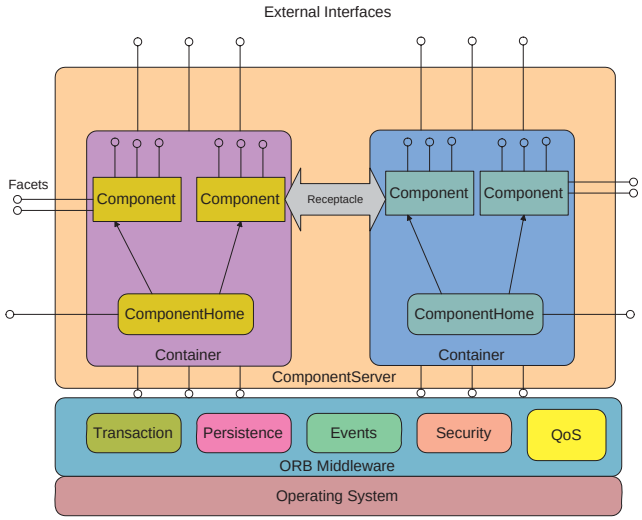

Figure 1.1: Key Elements in the CORBA Component Model

whereas components are designed to have a range of capabilities that enable their reuse in other contexts. As shown in Figure 1.1, some key characteristics of component middleware that help the development of complex enterprise distributed systems include:

- Support for transparent remote method invocations,
- Exposing multiple views of a single component,
- Language-independent component extensibility,
- High-level execution environments that provide layer(s) of reusable infrastructure middleware services (such as naming and discovery, event and notification, security and fault tolerance),
- Tools that enable application components to use the reusable middleware services in different compositions.

1.3 Overview of Research Challenges

While component middleware provide a number of advantages over previous technologies, several vexing problems remain. Some of the key challenges in developing, deploying and configuring component-based large-scale enterprise DRE systems using component middleware include:

1. **Lack of system composition tools.** While component middleware provides a lot of tools for developing individual components using general purpose programming languages, there are few tools that exist for

composing systems from individual components. Thus developers are still forced to deal with composition using previous generation tools like IDEs. Such tools lack the ability to check architectural constraints of the system and hence these problems don't show up until the system is deployed, or worse after the system has been deployed, *i.e.*, at run-time. Since it costs much more to fix problems later in the software development cycle, lack of system composition tools is a big challenge to ensure successful adoption of component middleware technologies. A key research challenge is therefore the lack of system composition tools that focus on strategic architectural issues, such as system-wide correctness and performance, and provide an integrated view of the system.

2. **Complexity of declarative platform API and notations.** Over the years, complexity of the platform API have evolved faster than the ability of general-purpose languages to mask this complexity. For example, popular middleware platforms, such as EJB and .NET, contain thousands of classes and methods with many intricate dependencies and subtle side effects that require considerable effort to program and tune properly. Though these platforms expose *declarative* techniques for performing various system development and deployment tasks, the technologies chosen to express these *declarative* techniques are often not user-friendly. For example, platforms like .NET, EJB and CCM use XML [16] technologies as the notation for all *metadata* related to specification of policy, configuration of middleware as well as deployment of applications. A key research challenge is therefore the complexity of declarative platform API and notations of the metadata prevalent in the component middleware technologies.

3. **Overhead due to high-level abstraction in large-scale systems.** In any complex system built using components, it is rare to find a single component that realizes a complex functionality on its own, *i.e.,* as a standalone component. We refer to indivisible, standalone components as *monolithic* components. Each monolithic component normally performs a single specific functionality to allow reuse of its implementation across the whole system. Thus, a number of inter-connected components are often composed together to create an assembly which realizes the complex functionality. Each such composition of components into an assembly results in a small and often unnoticeable overhead compared to implementing the functionality as a single component. By applying the same principle to composing the entire system, it is easy to get into a situation where a number of such small overheads add up to become a significant portion of the total execution time, thereby causing reduced QoS to clients. In worst cases, the application overhead can become so intolerable that the system is no longer usable. Thus a key research challenge is the composition overhead of the high-level component middleware technologies when applied to large-scale systems.

1.4 Research Approach

To address the problems with the complexity of platforms and the inability of third-generation languages to alleviate this complexity and express domain concepts, we propose an approach that applies Model-Driven Engineering (MDE) technologies to the design, development and deployment of component-based enterprise DRE systems. As shown in Figure 1.2, our approach involves a combination of:

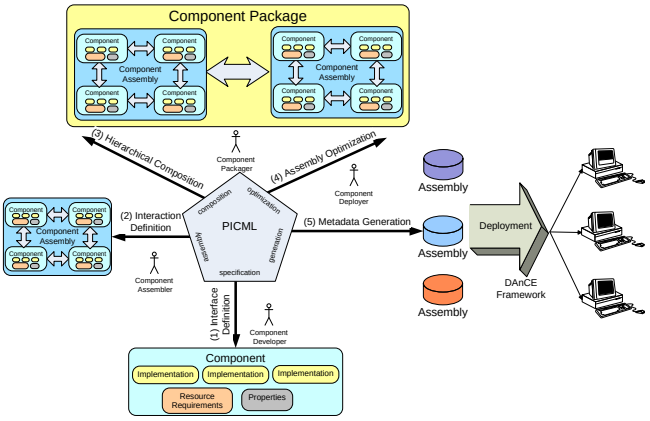

Figure 1.2: Research Approach

- ***System Composition Technologies***, which includes a domain-specific modeling language and associated tools to allow component interface definition, component interaction definition and multi-level composition of systems from individual components. The proposed research provides system level composition tools that allows composing systems from individual components. Section 2.3 describes the system composition tools in detail.

- ***Generative Technologies***, which includes a number of tools that utilize the system composition technologies outlined above to capture and automate the generation of metadata automatically from the models. By automating the generation of platform-specific metadata from models, the proposed research alleviates the problems with the complexity of the declarative notations. Section 3.3 describes the generative technologies in detail.

- ***System Optimization Technologies***, which includes an optimization framework that uses the application context available in the models to optimize the execution and footprint of applications built using components. By

performing optimizations that were previously infeasible to perform efficiently by operating at the middleware level, the proposed research optimizes away the composition overhead associated with component middleware technologies. Section 4.3 describes the optimization technologies in detail.

1.5 Proposal Organization

The remainder of this proposal is organized as follows: each chapter describes a single focus area, describes the related research, the unresolved challenges, our research approach to solve these challenges, and evaluation criteria for this research. Chapter 2 describes the issues related to composition of component systems, Chapter 3 deals with expression of design intent and Chapter 4 deals with application-specific optimizations. Finally, Chapter 5 provides a summary of the research contributions, publications and a time-line for the thesis proposal.

Chapter 2

Composition of Component Systems

System composition refers to composing a system by inter-connecting different individual components. Figure 2.1 shows the different dimensions across which component composition is defined. These include:

- **Structural Dimension.** Structural dimensions are related to the structural properties of composition of a system. Systems can be sub-divided into two categories structurally:
  1. *Flat*, where connections between the components in the system are at the same level; all the components are defined at the same level, *i.e.*, they are peers,
  2. *Hierarchical*, where components are grouped together into assemblies which may further be composed of sub-assemblies, and connections between components exist at both levels.
- **Temporal Dimension.** Temporal dimension is related to the time at which the composition happens. Systems can be sub-divided into two categories in the temporal dimension:
  1. *Static*, where the components are combined together at build time statically,
  2. *Dynamic*, where the connections between components are orchestrated at deployment time using declarative metadata by a deployment engine.

Component middleware promotes the development of libraries of pre-built and tested individual components, which offer different levels of capabilities and performance to clients. While this paradigm increases the opportunities for systematic reuse, it can also complicate software lifecycle processes. In

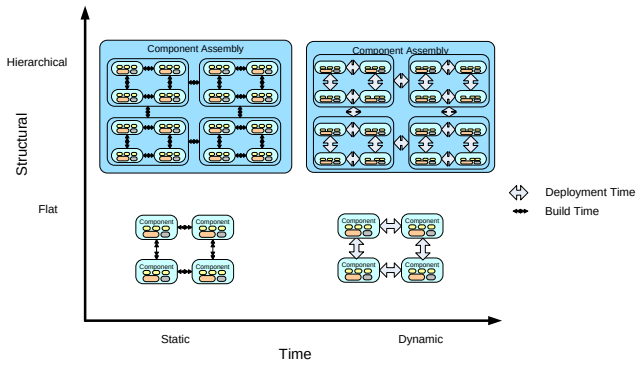

Figure 2.1: Composition Dimensions

particular, component middleware shifts responsibility from software development engineers to other types of software engineers (such as software configuration and deployment engineers) and systems engineers. Software development engineers traditionally created entire applications in-house using top-down design methods that could be evaluated throughout the lifecycle. In contrast, software configuration and deployment engineers and system engineers today must increasingly assemble enterprise, distributed systems by customizing and composing reusable components from existing frameworks, rather than building them from scratch. Thus it is clear that system composition is becoming a critical part of enterprise DRE system development.

2.1 Related Research

Composition of component-based systems has been studied extensively in the research community. Research on composition of component-based systems can be broadly categorized into three categories: (1) Component Development Environments, which deal with graphical environments that allow definition and composition of components, (2) Component Programming Techniques, which deals with improvements to programming languages and new programming methodologies and techniques to support component composition, (3) Functional Verification of Components, which deals with verification of components and compositions for various properties like QoS, deadlocks, real-time behavior. These three areas are discussed below:

1. **Component Development Environments.** The *Embedded Systems Modeling Language* (ESML) [17] was developed at the Institute for Software Integrated Systems (ISIS) to provide a visual metamodeling language based on GME that captures multiple views of embedded systems, allowing a

diagrammatic specification of complex models. The modeling building blocks include software components, component interactions, hardware configurations, and scheduling policies. Using these analyses, design decisions (such as component allocations to the target execution platform) can be performed. ESML is platform-specific since it is heavily tailored to the Boeing Boldstroke PRiSm QoS-enabled component model [18, 19]. ESML also does not support nested assemblies and the allocation of components are tied to processor boards, which is a proprietary feature of the Boldstroke component model.

Ptolemy II [20] is a tool-suite that supports heterogeneous modeling, simulation, and design of concurrent systems using an actor-oriented design. Actors are similar to components, but their interactions are controlled by the semantics of models of computation, such as discrete systems. The set of available actors is limited to the domains that are natively defined in Ptolemy. Using an actor specialization framework, code is generated for embedded systems. Ptolemy II supports components based on Java, with preliminary support for C.

WREN [21] is a component-based environment that emphasizes building systems composed of components retrieved from common software distribution sites as opposed to being completely developed in-house. The work also identifies some key requirements of component-based development environments including support for modular design, self-description, presence of global namespaces, support for application composition in addition to component development, support for component configuration, support for multiple views and reuse through reference to alleviate the maintenance problems.

2. **Component Programming Techniques.** A comprehensive collection of work related to Component-Based Software Engineering (CBSE) including definition of components, component-models and services, business case for components, product-line architectures, software architectures, standard-based component models as well as legal implications of component-based software is [22]. Research on composition techniques at the programming language level include the work on Scala [23], extensions to languages to support collaboration-based designs using mixin-layers in a static fashion [24] as well as in a dynamic fashion [25]. The topic of generating product-line architectures has been addressed in [26] with an extension of this work to non-code artifacts in [27]. A seminal work on defining generative programming methodologies, tools and applications is [28]. Other work on composition techniques include the work on variability management in the context of product-line architectures in [29], which compares Feature-Oriented Programming(FOP) [30] with Aspect-Oriented Programming(AOP) [31]. Recent efforts [32] have also been focused on optimal strategies for composition of Web Services, where a number of publically available Web Services are composed together to

satisfy a high-level requirement. A good summary of the existing techniques and requirements for composition of Web Services is [33].

3. **Functional Verification of Components.** Cadena [34] an integrated environment for building and modeling CCM systems. Cadena provides facilities for defining component types using CCM IDL, specifying dependency information and transition system semantics for these types, assembling systems from CCM components, visualizing various dependence relationships between components, specifying and verifying correctness properties of models of CCM systems derived from CCM IDL, component assembly information, and Cadena specifications, and producing CORBA stubs and skeletons implemented in Java.

   The *Virginia Embedded Systems Toolkit* (VEST) [35] and the *Automatic Integration of Reusable Embedded Systems* (AIRES) [36] are analysis tools that evaluate whether certain timing, memory, power, and cost constraints of real-time and embedded applications are satisfied. Components are selected from pre-defined libraries, annotations for desired real-time properties are added, the resulting code is mapped to a hardware platform, and real-time and schedulability analysis is done.

Much emphasis of the related research has been on component programming models and languages to allow construction of components, *i.e.*, how to write better components, and functional verification of individual components. Another issue with related research is that a lot of *tool-specific component technologies* have been proposed, whereas there is a need for *component technology agnostic tools*. However, with the standardization of component programming models, and the availability of commercial-off-the-shelf (COTS) components, focus needs to shift away from "programming-in-the-small" to "composing-systems-in-the-large", and away from proprietary component models to standards based component models. Another area which has not been given enough attention is the deployment of component-based systems and support for managing deployment artifacts. Section 2.2 describes the key unresolved challeges in composition of component-based systems, that forms the basis for our research.

2.2 System Composition: Unresolved Challenges

As shown in Figure 2.2, the challenges in building distributed systems are thus shifting from focusing on the construction of individual components to, composition of systems from a large number of individual subcomponents, and ensuring correct configuration of the subcomponents. Composition of systems from individual components needs to ensure that the connections between components are compatible, as well as ensure that the deployment descriptors for the composed systems are valid.

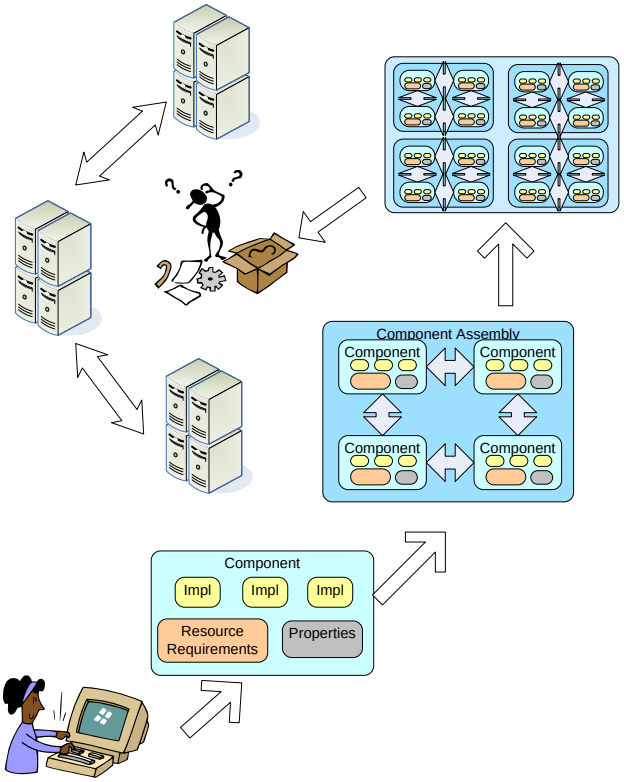

Figure 2.2: Compositions of Systems from COTS Components

Unfortunately, problems associated with composing systems from components often become manifest only during the integration phase. Problems discovered during integration are much more costly to fix than those discovered earlier in the lifecycle. A key research challenge is thus exposing these types of issues (which often have dependencies on components that are not available until late in development) earlier in the lifecycle, e.g., prior to the system integration phase. The following is a list of the unresolved challenges with composition of systems from standards based components:

1. **Lack of tool support for defining consistent component interactions.** Existing inteface definition tools are primitive in the sense that the interfaces for different components are specified separately, and getting the inteface definitions right involves tedious edit, compile, fix cycle. Also while the individual interfaces themselves may be strongly typed, the lack of component interconnection information in interface definitions languages like CORBA Interface Definition Language (IDL) [13] and Web

Services Definition Language (WSDL) [37] makes the task of composing systems more difficult. This is because inconsistencies in the component interactions are not detected until either deployment-time, or in some cases until run-time.

2. **Lack of integrated system view.** Traditional environments for component development provide a split view of the system where there is a design view, *e.g.*, Unified Modeling Language (UML) [38] models of the system, and there is a development centric view, *e.g.*, Microsoft Visual Studio, Eclipse [39]. Thus there is a lack of an integrated system view to help the system developers reason about systems at the system level.

3. **Lack of tool support for multi-level composition.** System developers also need tools that allow viewing the system at multiple levels of granularity (complexity). Also when one portion of a system changes, the change propagation is done in an *ad hoc* fashion which is tedious and error-prone when done manually.

4. **Lack of context-aware policy specification mechnanisms.** Components of a system might need to be configured with different parameters based on the usage context of a component. However, existing integrated development environments (IDEs) lack support for context-aware policy specifications which results in maintenance issues when the number of components and the number of contexts in which a component is used in a system grows.

5. **Lack of scalable composition techniques.** Most graphical environments and composition techniques are effective when the number of components in a system number in the tens or hundreds. However, when the number of components in a system is in thousands, it is extremely unproductive to perform composition activities manually. Even if a tool environment provides such a capability, it may not be customizable. Existing mechanisms to customizing an environment involve writing plugins, or addons [40], which assumes familiarity with the tool environment itself and is an extra burden on system developers.

Hypotheses This thesis proposes to build a system composition tool infrastructure which will explore and validate the following hypotheses with respect to system composition:

1. Disallow inconsistent component interactions
2. Provide integrated view of system
3. Support system composition across multiple levels
4. Support context-dependent policy specifications
5. Support scalable composition techniques

6. Support integration with component build tools and component repositories

2.3 Solution Approach → System Composition Tools

To address the unresolved challenges with system composition outlined in Section 2.2, we have developed the *Platform-Independent Component Modeling Language* (PICML) [41]. PICML is an open-source domain-specific modeling language (DSML) available for download at `http://www.dre.vanderbilt.edu/cosmic/` that enables developers of component-based DRE systems to define application interfaces, QoS parameters, and system software building rules, as well as generate valid XML descriptor files that enable automated system deployment as shown in Figure 2.3. PICML is developed using the Generic Modeling Environment (GME) [42], a meta-programmable domain-specific modeling environment. GME is a meta-programmable modeling environment with a general-purpose editing engine, separate view-controller GUI, and a configurable persistence engine. Since GME is meta-programmable, the same environment used to define PICML is also used to build models, which are instances of the PICML metamodel. PICML is the core of the CoSMIC [43] toolchain.

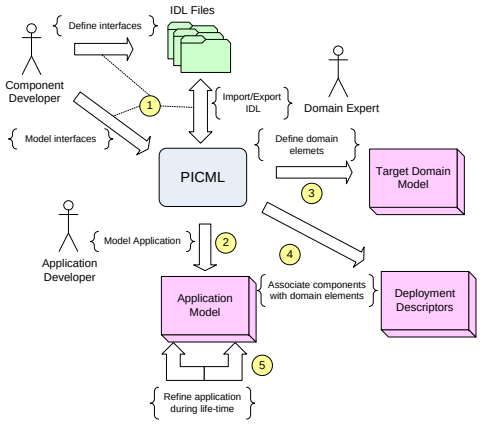

Figure 2.3: System Composition using PICML

At the core of PICML is a DSML (defined as a *metamodel* using GME) for describing components, types of allowed interconnections between components, and types of component metadata for deployment. The PICML metamodel defines ∼115 different types of basic elements, with 57 different types of associations between these elements, grouped under 14 different folders. PICML

allows system developers to capture the components and the dependencies in a visual fashion. PICML defines *static semantics* using OMG's Object Constraint Language [44]. *Dynamic semantics* are defined via model interpreters, a DLL that is loaded at run-time into GME and executed to perform various generative actions.

We now show how our research in building PICML, a system composition tool, validates the hypotheses presented in Section 2.2 as follows:

1. Since PICML [45] defines a metamodel that captures the abstract syntax of the CCM programming model, it is able to syntactically validate the component interaction definitions. PICML also ensures that the platform semantics, *i.e.*, CCM semantics are captured using OCL constraints. Thus, it is not possible to construct models using PICML that are either syntactically invalid or violate platform semantics.

2. PICML exploits the multi-aspect visualization capabilities of GME to allow the system developer to visualize the system according to different concerns including Interface Definition, Packaging, NodeMapping, QoS and ComponentMetrics, from within the same environment. Thus, PICML provides an integrated view of the system to the system developer.

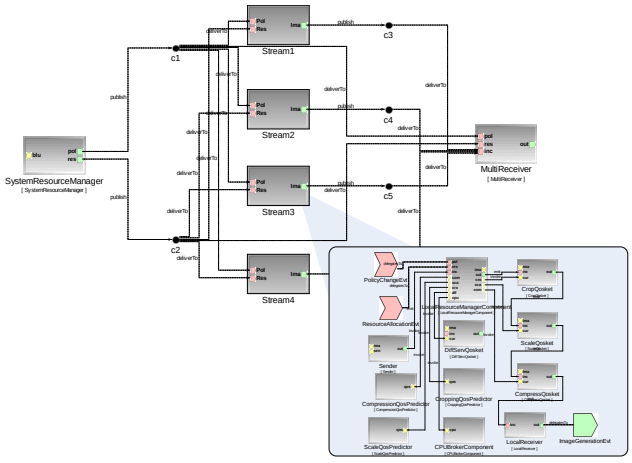

Figure 2.4: Hierarchical Composition Techniques

3. As shown in Figure 2.4, PICML allows composition across multiple levels of hierarchy. Using PICML a system developer can model components, compose component into component assemblies by interconnecting them, and reuse component assemblies into higher-level assemblies in an unrestricted fashion. Assemblies are defined as types in PICML.

This allows instantiation of assemblies multiple times, which is a major benefit since the changes to the composition of an assembly type automatically gets propagated to all the instances.

4. PICML also allows both components and assemblies to be associated with *properties*. Property elements can be specified at multiple levels in the hierarchy which allows a flexible mechanism for overriding values at any level. Policy specification can also be done at the assembly type level and overridden in the assembly instances. Thus, PICML allows context-dependent policy specifications.

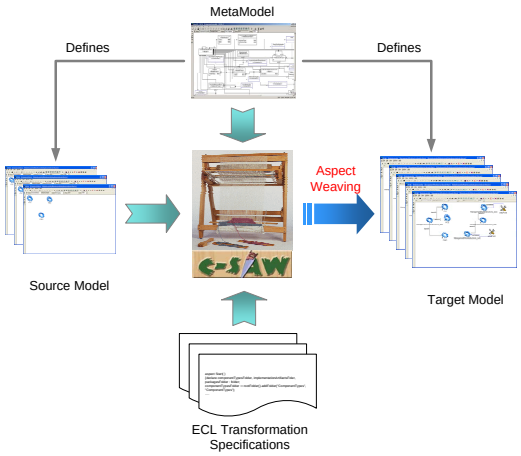

Figure 2.5: Scalable Composition Techniques

5. As shown in Figure 2.5, PICML also integrates support for scalable composition techniques [46]. This work was done in colloboration with the team from Software Composition and Modeling Laboratory (SOFTCOM). PICML uses Constraint-Specification Aspect Weaver (C-SAW) [47], an aspect-oriented model weaver to weave in crosscutting concerns [48], *i.e.*, composition, into model elements. Specification of constraint to be used in weaving is given as an input to the C-SAW tool. By automating the weaving of crosscutting concerns using C-SAW, PICML is able to provide an environment that is both customizable (what elements to weave and according to what constraints) as well as scalable (how many elements to create, where and how many times to repeat).

2.4 Proposed Enhancements

To validate the last hypothesis (support integration with component build tools and component repositories), we propose to enhance PICML to automatically

generate software build rules from models. To be able to build components in an operating system independent fashion, we propose to utilize a tool called MakeFile, Project and Workspace Creator (MPC) [49], which generates software build artifacts like Makefiles, Microsoft Visual Studio solution files necessary to build a component system. In addition to generating plain build files, we also propose to exploit the application context available at the modeling level to optimize the building of components. For example, with knowledge about the interfaces of a component that actually take part in inter-connections, it is possible to achieve footprint reductions by restricting the visibility of symbols exposed from a component, to the ones that are actually used. This technique take advantage of the native platform build tool capabilities to perform this optimization.

In order to ensure that the system developers deal with implementation artifacts associated with a component at a higher-level of abstraction, it is necessary to provide support for managing components in terms of *packages*. We propose to build an infrastructure that automates the packaging of components and also pushes the packages to a component repository. This enables run-time infrastructure to get access to the metadata thereby opening up new avenues for run-time reflection.

2.5 Evaluation Criteria

To validate the hypothesis we propose that the enhancements be compared against the following baseline:

Compared with the deployment of a system with existing tools, the proposed enhancements should:

1. Eliminate the need to manually write build scripts.
2. Reduce static footprint per component type by 20%. Preliminary testing already give a 10% reduction per component type. The total savings in a large system with a number of unique component types will be much higher.
3. Automate the creation of component packages.
4. Automatically deploy a package into a component repository.

Chapter 3

Expression Of Design Intent

Prior generations of middleware, such as Remote Procedure Calls (RPC) and Distributed Object Computing (DOC), used tightly coupled imperative techniques throughout the development, deployment, and configuration of system artifacts. As a result, most application and platform code is still written and maintained manually using third-generation languages, which incurs excessive time and effort particularly for key integration-related activities, such as system deployment, configuration, and quality assurance. For example, it is hard to write Java or C# code that correctly and optimally deploys large-scale distributed systems with hundreds or thousands of interconnected software components. A key culprit is the significant semantic gap between design intent (such as deploy components 1-50 onto nodes A-G and components 51-100 onto nodes H-N in accordance with system resource requirements and availability) and the expression of this intent in thousands of lines of handcrafted third-generation languages. To address the deficiencies with imperative solutions expressed in third-generation languages, component middleware provides a significantly richer declarative notation for codifying various development, deployment, and configuration activities. Common examples of such declarative notations include:

- Usage of XML descriptors to configure application component properties (such as initial values for component attributes) at deployment time.
- Usage of XML descriptors to configure critical pieces of the middleware infrastructure, such as event-de-multiplexing mechanisms, level of concurrency.
- Contents of component packages (as opposed to platform-specific shared libraries or dynamically linked libraries) along with meta-information to represent component implementations.
- Usage of XML descriptors to orchestrate the deployment of systems, e.g., the number of instances of each component type and the connections between component instances.

3.1 Related Research

Declarative notations to express design intent is a hot-topic in the research community. Manifestations of the declarative approach to configuration of the system starts from operating system level and goes all the way up to component-based system packaging. We explore the related research in utilizing declarative notations under four categories:

1. **Operating System.** Research on administration of personal computer systems [50] has focused on replacing the imperative updates to configuring and updating the operating system with declarative techniques, which rely on a system model as a function that can be applied to a collection of system parameters to produce a statically typed, fully configured system instance. This research has been prototyped on Singularity OS.

   On the other end of the spectrum is Pan [51], a high-level configuration language for system administration of a large number of machines, ranging from large clusters to desktops in large organizations. The approach taken to configuration is to store configuration information in a database in two alternate forms: a high-level declarative description (Pan) and a low-level XML based notation. Automated tools are provided which convert Pan to XML based format.

   Declarative notations have also been applied to the task of instrumenting a live system as implemented in DTrace [52]. DTrace is an online instrumentation facility which uses a declarative high-level language to describe predicates and actions at a given point of instrumentation. The DTrace mechanism has been integrated into the Solaris Operating system.

2. **Software Architecture Description.** Emerging standards like Web Services are based on Web Services Description Language [37] to describe Web services starting with the messages that are exchanged between the service provider and requestor. The messages themselves are described abstractly and then bound to a concrete network protocol and message format. A message consists of a collection of typed data items. An exchange of messages between the service provider and requestor are described as an operation. WSDL uses XML Schema [53,54] as the language for describing the service descriptions.

   xADL [55] is an infrastructure for development of software architecture description languages (ADLs), which relies on using XML for description of the language itself. It provides a base set of reusable and customizable architectural modeling constructs and an XML-based modular extension mechanism. Primary goal of xADL is to unify the plethora of ADL notations in prevalence, and to reduce the effort expended in building tools to support ADLs.

3. **Middleware Communication.** Recent research on network protocol design [56] has resulted in a generic application protocol kernel for connection-oriented, asynchronous interactions called BEEP. Messages are usually textual (structured using XML). BEEP is itself not a protocol for sending and receiving data directly. Rather, it allows definition of application protocol in a declarative fashion on top of it, reusing several mechanisms such as: asynchronous communications, transport layer security, peer authentication, channel multiplexing on the same connection, message framing, and channel bandwidth management.

   Another declarative RPC protocol that is becoming popular is the Simple Object Access Protocol (SOAP) [57]. SOAP provides a simple and lighteweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML. SOAP does not itself define any application semantics such as a programming model or implementation specific semantics; rather it defines a simple mechanism for expressing application semantics by providing a modular packaging model and encoding mechanisms for encoding data within modules. This allows SOAP to be used in a large variety of systems ranging from messaging systems to RPC.

4. **Middleware Packaging.** The .NET framework employs a number of declarative mechanisms to build, configure and deploy [58] systems. The use of declarative notations (built on top of XML) is pervasive in the .NET architecture. XML is used to configure the run-time behavior of not just shared libraries (assembly in .NET parlance) but also entire applications. XML is also used to describe the configuration of security policies at various levels of abstraction including application, machine or even enterprise. the next generation

   Other standards-based component middleware including CCM [13] and EJB [11] also define declarative mechanisms using XML for compostion and assembly of components and component packages, as well as for orchestrating deployment of component systems [59].

While the move towards declarative notations is an advance over previous generation imperative techniques, the declarative techniques have chosen to use tool-friendly technologies like XML as the medium for expression of design intent. XML is non-intuitive and error-prone to write manually (with or without tool support). Any changes to a system requires modification to XML which is a cumbersome process. Thus by choosing XML as the underlying notation for declarative techniques, the problems associated with imperative system configuration have just been shifted into a different space, *i.e.*, configuring using XML.

3.2 Expression of Design Intent: Unresolved Challenges

The use of declarative notations in component middleware automates hard-coded configuration and deployment activities and increases the reuse of components by eliminating tedious and error-prone manual configuration.

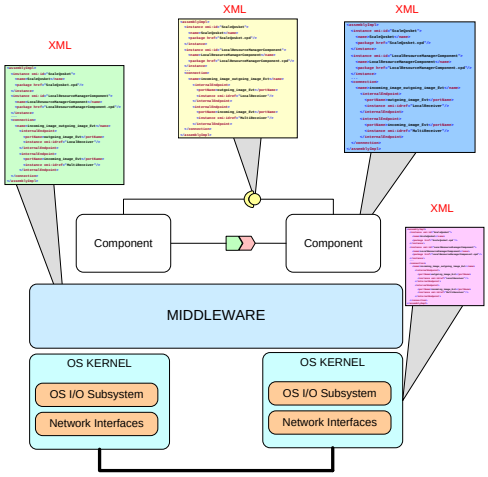

Figure 3.1: Declarative Notations in Component Middleware

However, as shown in Figure 3.1, expression of these declarative notations in textual languages like XML merely shift the burden of the component developers from writing code to writing the configuration and deployment descriptors. For example, there is still a significant semantic gap between the design intent (e.g., deploy components 1-50 onto nodes A-G and components 51-100 onto nodes H-N in accordance with system resource requirements and availability) and the expression of this intent in thousands of lines of hand-crafted XML, whose visually dense syntax conveys neither domain semantics nor design intent. As a result, the use of text-based declarative notations has the potential to overwhelm component developers without adequate tool support. The following is a list of unresolved challenges with respect to expression of design intent when using standards-based component middleware:

1. **Complexity of Declarative Notations.** A key research challenge is the accidental complexities of the declarative notations required to configure the component middleware. There are several examples which illustrate the problems with accidental complexities. For example, IDL for CCM (*i.e.*, CORBA 3.x IDL) defines extensions to the syntax and semantics of CORBA 2.x IDL. Every developer of CCM-based applications must therefore master the differences between CORBA 2.x IDL and CORBA

3.x IDL. For example, while CORBA 2.x interfaces can have multiple inheritance, CCM components can have only a single parent, so equivalent units of composition (*i.e.*, interfaces in CORBA 2.x and components in CCM) can have subtle semantic differences. Moreover, any component interface that needs to be accessed by component-unaware CORBA clients should be defined as a *supported* interface as opposed to a *provided* interface. In any system that transitions from an object-based architecture to a component-based architecture, there is likelihood of simultaneous existence of simple CORBA objects and more sophisticated CCM components. Design of component interfaces must therefore be done with extra care.

Another example from .NET related to naming is with the manifest files associated with an assembly. Each assembly in .NET (equivalent to a component in CCM) is associated with a manifest file. A manifest file is an XML file that can be used to configure the behavior of the assembly. Before the .NET run-time loads an assembly, it searches for any manifest files associated with the assembly. This search is designed to look for manifest files that follow a specific naming convention, and only in specific directories: For each assembly implemented as a shared library, say Foo.dll, the .NET run-time searches for a manifest file named Foo.dll.manifest; for each assembly implemented as an executable, say Foo.exe, the .NET run-time searches for a manifest file named Foo.exe.manifest. If the manifest files are given different names, the .NET run-time does not load the manifest, which might result in a different behavior for the assembly Foo.

2. **Lack of support for system evolution.** Another research challenge is maintaining and evolving the declarative metadata associated with a system. Any complex system will undergo a number of changes (minor and major) as part of it's evolution, and hence it is critical that the declarative metadata also evolve with the system. *Ad hoc* and *naive* approaches to management of metadata will result in problems during deployment time, or even at run-time, both of which are costly to fix.

Hypotheses This thesis proposes to build a generative tool tool infrastructure which will explore and validate the following hypotheses with respect to metadata management:

1. Provide high-level abstractions for specification of design intent
2. Automate generation of platform-specific metadata from these abstractions
3. Provide automated support system interface and metadata evolution
4. Provide support for developing heterogeneous systems, *i.e.*, systems which are built using combinations of multiple middleware platforms like CCM and .NET Web Services.

3.3 Solution Approach → Model-driven Generation

Models are effective vehicles for representing the declarative notations pervasive among the current generation of component middleware technologies. Models provide a high-level abstraction, which shields the component developers from the accidental complexities of the declarative notations. For example, models allow the developers to use model elements to specify the connections between components, but translate them into the different declarative notations required by different component middleware. It also allows the developer to focus on the problems one item at a time. Thus, the developer can specify the system resource requirements or availability requirements separately from the deployment requirements such as deploy components 1-50 onto nodes A-G and components 51-100 onto nodes H-N. From these requirements, the modeling infrastructure can automatically determine if the deployment is valid by evaluating the constraints.

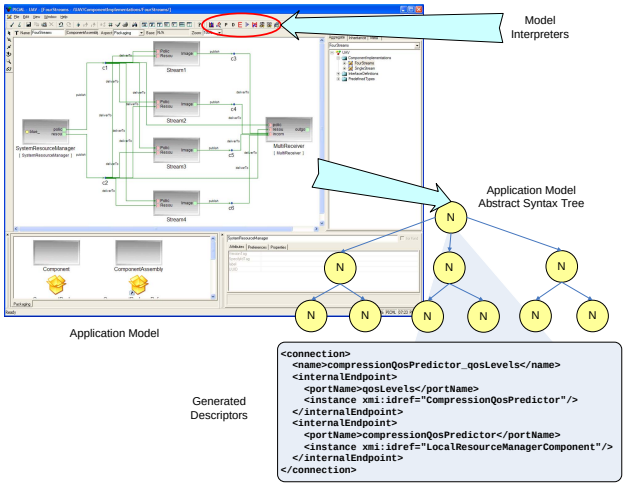

Figure 3.2: Automated Generation of Declarative Notations

Using GME tools, the PICML metamodel can be compiled into a *modeling paradigm*, which defines a domain-specific modeling environment. From this metamodel, ∼20,000 lines of C++ code (which represents the modeling language elements as equivalent C++ types) is generated. This generated code allows manipulation of modeling elements, *i.e.*, instances of the language types using C++, and forms the basis for writing *model interpreters*, which traverse the model hierarchy to perform various kinds of generative actions, such as generating XML-based deployment plan descriptors. PICML currently has ∼8 interpreters using ∼222 generated C++ classes and ∼8,000 lines of hand-written C++ code that traverse models to generate the XML deployment descriptors needed to support the OMG Deployment and Configuration speci-

fication [59]. Each interpreter is written as a DLL that is loaded at run-time into GME and executed to generate the XML descriptors based on models developed by the component developers using PICML. Examples of the type of descriptors generated by PICML which aid in deployment of a system built using OMG's Deployment and Configuration Specification include:

- **Component Interface Descriptor (.ccd)**, which describes the interfaces ports, attributes of a single component.
- **Implementation Artifact Descriptor (.iad)**, which describes the implementation artifacts (e.g., DLLs, executables etc.) of a single component.
- **Component Implementation Descriptor (.cid)**, which describes a specific implementation of a component interface; also contains component inter-connection information.
- **Component Package Descriptor (.cpd)**, which describes multiple alternative implementations (e.g., for different OSs) of a single component.
- **Package Configuration Descriptor (.pcd)**, which describes a component package configured for a particular requirement.
- **Component Deployment Plan (.cdp)**, which guides the run-time deployment.
- **Component Domain Descriptor (.cdd)**, which describes the deployment target, i.e., nodes, networks on which the components are to be deployed.

As shown in Figure 3.2, our research in automating the generation of metadata [60] in the context of PICML resulted in capabilities, which enable developers of component-based systems to define application interface, optional parameters, system software build rules, and generates valid XML descriptor files that enable automated system deployment. In essence, PICML defines a type system that includes the entities of CCM metadata as first-class objects. This type system can be used to validate deployment and configuration models and automatically generate descriptors necessary to configure and deploy a system.

Our research also resulted in a set of component, interface, and other datatype definitions to be created using either of the following approaches:

- **Adding to existing definitions imported from IDL**. In this approach, existing CORBA software systems can be easily migrated to take advantage of the composition and generative technologies provided by PICML using its *IDL Importer*, which takes any number of CORBA IDL files as input, maps their contents to the appropriate PICML model elements, and generates a single XML file that can be imported into GME as a PICML model. This model can then be used as a starting point for modeling assemblies and generating deployment descriptors.

- **Creating IDL definitions from scratch**. In this approach, PICML's graphical modeling environment provides support for designing the interfaces using an intuitive "drag and drop" technique, making this process largely self-explanatory and independent of platform-specific technical knowledge. Most of the grammatical details are implicit in the visual language, *e.g.*, when the model editor screen is showing the "scope" of a definition, only icons representing legal members of that scope will be available for dragging and dropping.

CORBA IDL can be generated from PICML, enabling generation of software artifacts in languages having a CORBA IDL mapping. For each logically separate definition in PICML, the generated IDL is also split into logical file-type units. PICML's interpreter will translate these units into actual IDL files with #include statements based on the inter-dependencies of the units detected by the interpreter. PICML's interpreter will also detect requirements for the inclusion of canonical CORBA IDL files and generate them as necessary. In addition to the above mentioned capabilities, we have also built infrastructure such that changes to system interface definitions (outside the model; in IDL files) after an initial model has been created can be automatically imported into the application model. Thus both (re-)import as well as (re-)export of interfaces are supported, which is critical to ensure smooth evolution of a system during development.

We now show how our research in managing metadata using PICML, validates the hypotheses presented in Section 3.2 as follows:

1. PICML provides a visual medium, i.e., visual models to express the design intent of the developer, which is a higher level of abstraction than the declarative notations of the component middleware. Thus, it is easy to see that the level of abstraction provided, *i.e.*, user-friendly drag and drop operations driven using a GUI, is at a sufficiently higher level of abstration than writing XML descriptors manually.

2. From the visual models of the component applications, PICML also generates the descriptors using the abstract syntax tree of the model, thereby relieving the developer from both learning the declarative notations used in the descriptors as well as ensuring that the descriptors are valid. Thus, PICML validates the hypothesis for automated generation of metadata.

3. Since PICML allow seamless (re-)import and (re-)export of system interfaces, and captures metadata as first-class objects in the model, it is easy to see that it support both system interface and metadata evolution in a systematic fashion.

3.4 Proposed Enhancements

To validate the last hypothesis (Provide support for developing heterogeneous systems, *i.e.*, systems which are built using combinations of multiple middle-

ware platforms like CCM and .NET Web Services), we propose to extend the generative capabilities of PICML to support the declarative notations used in the deployment of systems built using Microsoft .NET Web Services. This involves identifying the key elements of the .NET Web Services framework needed for deployment of systems. As shown in Figure 3.3, the enhanced PICML will generate the following types of .NET metadata elements:

- **Assembly manifests (.dll.manifest)**, which describe the contents of a single .NET component, version information, information about referenced assemblies, and dependencies on runtime environment.
- **Application manifests (.exe.manifest)**, which describes the metadata associated with an application (system) and is quite similar to the assembly manifests, except that it defines metadata for an entire application.
- **Application Configuration Files (app.config)**, which describes the various configuration elements needed for configuring an application's behavior.
- **Web Service Configuration File (web.config)**, which describes the various configuration elements needed for configuring externally facing subsystems of a complex, distributed web Service.
- **Policy configuration files (enterprisesec, machine, security.config)**, which describes the application specific policy elements including security that need to replace/be merged with the existing policies at multiple levels of control, i.e., enterprise level, machine level.

All these different configuration files use XML format and each configuration file type above is different from the other file types in subtle ways. As a result, component developers must master multiple different formats to ensure successful deployment of a system built using .NET, which is tedious and errorprone since the complexity and number of files increases with the complexity of the system. By capturing the elements of .NET Web Service framework at the modeling level, and automating the generation of the different descriptors, our proposed approach therefore resolves numerous accidental complexities associated with manually authoring the declarative notation embodied in the different descriptors. In order to support development of heterogeneous systems, it is also necessary to provide wrappers which automatically transform calls from one middleware platform to another transparently by acting as a bridge. As part of the proposed enhancement, we also plan to automatically generate an instantiation of such a prototype framework to enable systems that support calls between CCM and .NET Web Services.

3.5 Evaluation Criteria

As shown in Figure 3.4, to validate the hypothesis we propose that the enhancements be compared against the following baseline:

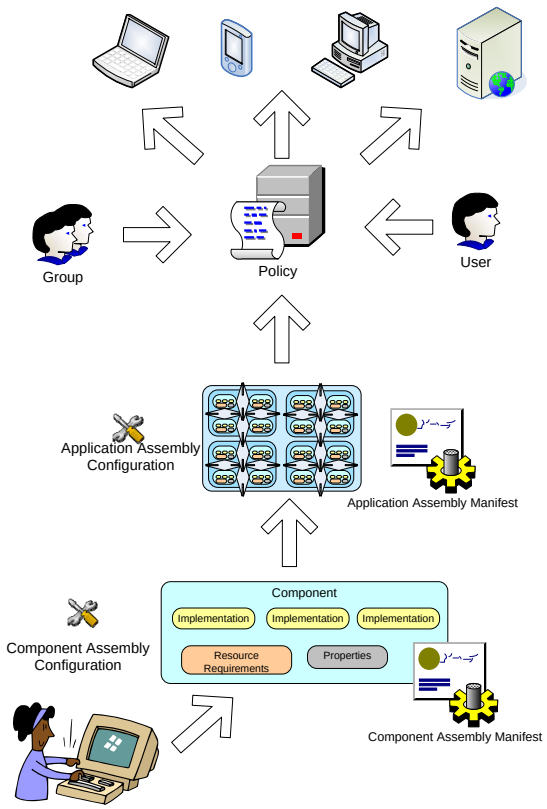

Figure 3.3: Generation of metadata for .NET Web Services

Compared with the deployment of a system with existing tools (vanilla Microsoft Visual Studio), the proposed enhancements should:

1. Automate a three-staged deployment — Development, Testing, & Production — of typical .NET Web services, *i.e.*, generate syntactically valid manifests for the three stages of deployment
2. Eliminate errors in deployment due to inconsistent Code-Access Security Policies
3. Eliminate need for user to write glue code to integrate a CCM component with a .NET Web Service.

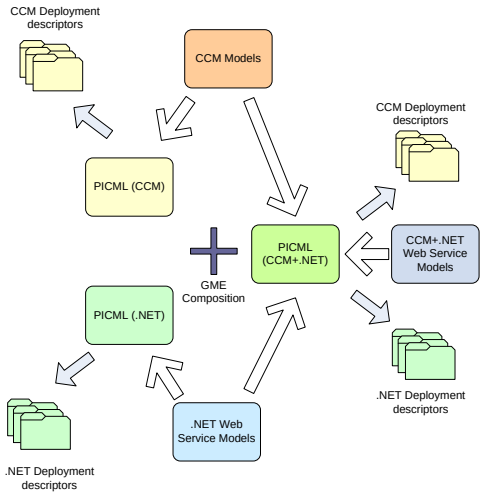

Figure 3.4: Development of Heterogeneous Component Systems

Chapter 4

Application Specific Optimizations

Over the past five decades, software researchers and developers have been creating abstractions that (1) help them to program in terms of their design intent rather than in terms of the underlying computing en- vironments (e.g., CPU, memory, and network devices) and (2) shield them from the complexities of these environments. From the early days of computing, these abstractions included both language and platform technologies. For example, early programming languages, such as assembly and FORTRAN, shielded developers from complexities of programming with machine code. Likewise, early operating system platforms, such as OS/360 and UNIX, shielded developers from complexities of programming directly to hardware. More recently, higher-level languages (such as C++, Java, and C#) and platforms (such as component middleware) have further shielded application developers from the complexities of the hardware. Although existing languages and platforms raised the level of abstraction, they can also incur additional overhead. For example, common sources of overhead in component middleware include marshaling/demarshaling costs, data copying and memory management, static footprint overhead due to presence of code paths to deal with every possible use cases, dynamic footprint overhead due to redundant run-time infrastructure helper objects, the endpoint and request de-multiplexing, and context switching and synchronization overhead. While some implementations of component middleware try to minimize this overhead, there is a limit to the optimizations done by the middleware developers. In particular, middleware developers can only apply optimizations that are applicable across all applications in a particular domain, which effectively limits the number of valid optimizations performed by default.

4.1 Related Research

Optimizing the middleware to increase the performance of applications has long been a goal of middleware system researchers. In this section we will explore a representative sample of the research that has been applied to optimizing middleware for component-based systems under two categories:

1. **Component Middleware** Research on optimizing component middleware techniques have relied on reflection techniques [61] to optimize away the overhead of invocations. Approaches to optimizations have focused on selecting optimal communication mechnanisms, managing QoS properties of component using the containers and dynamically (re)configuring selected portions of the component implementations. Other research has also focused on specialization of middleware for particular product-line architecture scenarios [62], utilize micro-ORB architectures [63–65].

   Other research [66] on optimizing component middleware have focused on effectively reusing legacy code along with component middleware technologies. The approach taken here is to separate the implementation of the business logic from the glue code necessary to implement CORBA object semantics. By taking advantage of an adapter layer and configuring the adapter to be local or remote, collocated invocations can be optimized to ordinary C++ function calls.

   Research on alternate component middleware like EJB have focused on automating the performance management [67] of applications by employing a performance monitoring framework which works in collaboration performance anomaly detection framework. By relying on redundant implementation of components, *i.e.*, component with same functionality but optimized for different run-time environments, implementations can be swapped for more optimal ones depending on the anomalies detected. The management framework also exhibits learning capabilities so that deployment time definition of good vs. bad performance is unnecessary.

   While most research on optimizing component middleware has focused on performing optimizations at the middleware layer, others [68] have focused on improving algorithms for event ordering within component middleware by making use of application context information available in models.

2. **Web Services** Research on optimizing Web Services in application specific fashion has focused on application specific data replication for edge services, *i.e.*, replicating servers at geographically distributed sites. By relaxing the consistency of data that is replicated in at the edge servers using application specific semantics [69] significant performance improvements in the latency and availability has been achieved.

   Other research on optimizing web services has focused on utilizing reflective techniques encapsulated in the request metadata [70] for dynamic

negotiation of best communication mechanisms between any requestor and provider of a service.

One common theme with the research on middleware optimizations has been the use of run-time reflection to adapt the behaviour of the middleware such that application performance is optimized. While this may be suitable for some system, not all enterprise DRE systems can afford the luxury of run-time reflection in the critical path. Another theme with the research on optimizations is the requirement for multiple implementations to be provided to the middleware to choose from. This strategy is not entirely application transparent, and imposes extra burden on the system developers. Finally, one of the important missing piece in the optimization research is the lack of a high-level notation to guide the optimization frameworks, *i.e.*, there is no intermediate abstract syntax tree (AST) of the application that is available to the middleware to use as a basis for performing optimizations.

4.2 Application Specific Optimizations: Unresolved Challenges

One of the biggest factors in affecting system performance is not a single significant decrease (which are usually easy to identify quickly) but a slew of small decreases. It is hard to notice this overhead creep into the system without a sophisticated Distributed Continuous Quality Assurance [71] infrastructure, and a considerable diligence on part of the developers, which does not scale up well to large-scale systems. Component middleware standards do not advocate any standard optimizations since it is not possible to perform them in the middleware without the knowledge of application context, i.e., such optimizations are not domain invariants. Tools that automatically optimize component assemblies (compositions) are not prevalent. It is hard to both identify and optimize component implementations manually, since the usage of components tends to span multiple hierarchies in any complex system. Further, an optimization that is applicable in one context may not be applicable in another context. Thus, it is not possible to perform these optimizations in isolation, but rather one should perform them based on every unique use-case. Finally, performing these optimizations manually by hand becomes infeasible with system evolution.

The following is a list of unresolved challenges with respect to application specific optimizations when using standards-based component middleware:

1. **Lack of application context.** A significant problem with component middleware is the number of missed optimization opportunities in the middleware due to lack of application context information. For example, when component middleware generates glue code to facilitate remote method invocations, it generates code with the assumptions that every

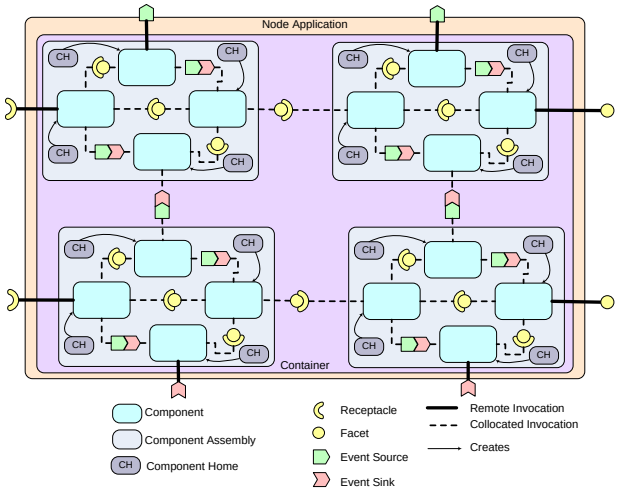

Figure 4.1: Composition Overhead in Component Assemblies

component is remote as shown in Figure 4.1. Often, however, all components that make up a subsystem are deployed onto the same node or even in the same process. Since the application composition information is not available during glue code generation, the middleware generated glue code is often inefficient because the glue code for local communication is much faster and smaller than glue code for remote communication. Although some implementations try to optimize the collocated invocations by generating two versions of the glue code, i.e., remote as well as local, it is still suboptimal since it increases memory footprint and decreased performance due to the check for collocation performed every time. Thus, it is impossible to solve this problem efficiently by operating at the middleware level only. A key research challenge is therefore to eliminate the overhead of applying high-level abstractions like component middleware *automatically* to ensure that the system still meets the desired performance requirements.

2. **Overhead of platform mappings.** Platform mappings for component middleware are typically defined with the assumption that every component is (or can be) remote. However, in certain cases, blind adherence to the platform mapping can result in significant overhead for applications built using component middleware. For example, in Figure 4.1 it is clear that the components that are internal to the assembly do not have any connection with other components outside. However, a default implementation of the internal components will generate a single factory object per component instance which is responsible for creation of components. This is wasteful in terms of both static (for each component type) and dynamic (for each component instance) footprint. A key re-

search challenge is to recognize such anomalies and optimize away the overhead.

3. **Lack of physical assembly mapping.** Standard mappings for popular component middleware lack the concept of a *physical* assembly mapping, and define only *virtual* assembly mappings. This imposes extra overhead at both deployment time (making all the necessary inter-component connections) and run-time (checked collocated method invocations) which is unnecessary when all the components are deployed onto a single node (and process). A key research challenge is to devise a physical mapping for an assembly.

Hypotheses This thesis proposes to build a system optimizer framework which will explore and validate the following hypotheses with respect to system composition optimization:

1. Supply application context available in system models to component middleware to perform optimizations
2. Optimize platform mappings using application context
3. Devise a mapping for physical component assembly and perform optimizations using this mapping

4.3 Proposed Approach → System Composition Optimizer

We propose to build an automatic system composition optimizer that relies on the availability of the application structure as a PICML model, and operates at a higher level of abstraction than traditional component middleware. The optimizer is able to derive the same application context(s) as can be derived by parsing the implementation source code of all components, which make up an assembly.

By maintaining a list of applicable optimizations, and checking each usage of each component within an assembly, as well as inter-assembly interactions, the optimizer can come up with a list of valid optimizations for every instance of an assembly. The optimizer feeds this information to the middleware by a combination of glue-code generation that utilizes the application context, as well as automating the generation of *smart* build rules to force such code generation. For example, the generated client-side proxy code for a CCM component has checks in the middleware to determine if the component that is the target of a method invocation is remote/local. However, if it can be determined from a model, that all components of an assembly are local, then the code that is generated for remote case in the component implementations can be eliminated as shown in Figure 4.2.

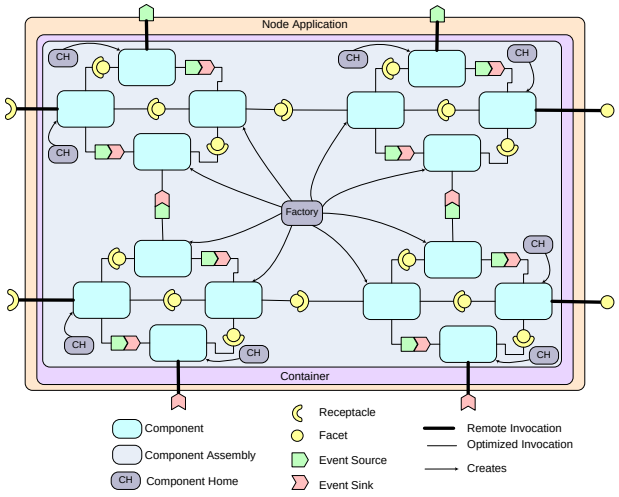

Figure 4.2: Optimized Component Assemblies

We propose to identify a list of such optimizations that are applicable to both CCM as well as .NET web services, and equip the optimizer to perform these optimizations upon user direction. Examples of sources of overhead eliminated include combining multiple component homes (factory component generated for every component) into a single component, transformation of a dynamic composition into a static composition, and reducing the context information maintained by the middleware corresponding to every component instance. Some optimizations like physical assembly mapping can be performed at multiple levels in the hierarchy as shown in Figure 4.3. In such cases, the optimizer will empirically evaluate the best depth in hierarchy to limit the optimization. The optimizer will perform all the optimizations possible without requiring modifications to the individual component implementations. Thus, these optimizations are completely transparent to the component developer, and there is no need to develop multiple alternate implementations of the same component(s).

4.4 Evaluation Criteria

To validate the hypothesis we propose that the enhancements be compared against the following baseline:

Compared with the performance and footprint of multiple applications such as, the Emergency Response System, ARMS GateTest scenarios, scenarios that exhibit inherent hierarchy as well as ones that don't have inherent hierarchy, the proposed enhancements should:

1. Reduce static and dynamic footprint. Given n components that are inter-

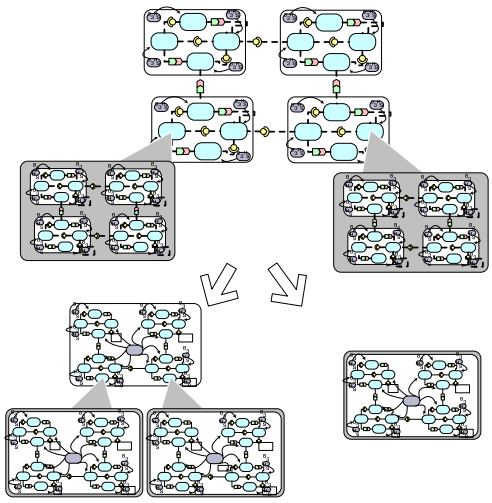

Figure 4.3: Component Assembly Fusion at Multiple Levels

nal to an assembly with x components in total, the no. of homes shold be reduced by (n-1)/x, the no. of components registered with the POA by (n-1)/x.

2. Increase the performance. Given t as the no. of interactions between components within an assembly, transform t collocation checked calls to t local calls

3. Eliminate mis-optimizations. All optimizations performed shouldn't violate platform and application semantics like incompatible policy, real-time QoS requirements.

4. Do not require changes to the component implementation

5. Optimization infrastructure is customizable and application transparent.

6. Investigate the feasibility of applying optimizations to .NET Web Services (in addition to CCM).

Chapter 5

Concluding Remarks

Component middleware is an emerging paradigm. Success of component middleware is crucial to realizing the vision of Software Factories [72]. However, there are significant gaps in the component middleware development and integration toolchain, which if left unresolved has the potential to negate benefits of using component middleware. Also, standards-based component middleware may not be suitable for direct application to build DRE systems. Our research which applied MDE technologies has resulted in improved tool-support for component middleware as shown in Table 5.1 and Table 5.2. As shown in Figure 5.1, this thesis proposes to further the benefits of applying MDE technologies by performing optimizations which exploit application context that were previously infeasible to perform in the middleware, or manually.

Table 5.1: Summary Of Research Contributions

Category	**Benefits**
System Composition Technologies	Support static composition of systems by ensuring that the components get built correctly; support dynamic composition of systems by ensuring that the connections between components are correct
Generative Technologies	Expressing domain constraints in the form of a DSML and automating the generation of syntactically valid metadata; reduces requirements of domain experts to also be implementation platform experts
System Optimization Technologies	Automatic discovery and realization of optimizations from models using application context; such optimizations are impossible to perform if operating at the middleware layer alone. Propose a novel mechanism for mapping an assembly as a component without any extra overhead

Table 5.2: Summary of Publications

Category	Publications
System Composition Technologies	1. Towards Composable Distributed Real-time and Embedded Software, Proceedings of the 8th IEEE Workshop on Object-oriented Real-time Dependable Systems (WORDS), Guadalajara, Mexico, January 2003. 2. Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems, the International Journal of Embedded Systems, special issue on Design and Verification of Real-Time Embedded Software, April 2005. 3. A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems, Elsevier Journal of Computer and System Sciences, 2006. 4. Weaving Deployment Aspects into Domain-Specific Models, International Journal on Software Engineering and Knowledge Engineering, Summer 2006 (accepted).
Generative System Technologies	1. A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems, 11th IEEE Real-Time and Embedded Technology and Applications Symposium, San Francisco, CA, March 2005. 2. Developing Applications Using Model-Driven Design Environments, Computer, vol. 39, no. 2, pp. 33-40, Feb., 2006. 3. Model-driven Development of Component-based Distributed Real-time and Embedded Systems, Model Driven Engineering for Distributed Real-time and Embedded Systems, Hermes, 2005. 4. Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications, Elsevier Journal of Science of Computer Programming: Special Issue on Model Driven Architecture, 2006.

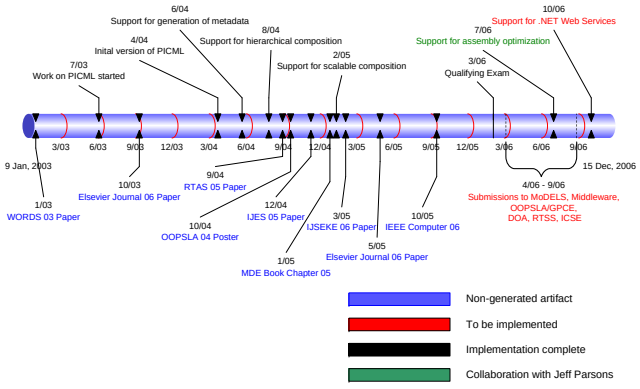

Figure 5.1: Doctoral Research and Dissertation Timeline

Bibliography

[1] B. Stroustrup, *The C++ Programming Language, Special Edition*. Boston: Addison-Wesley, 2000.

[2] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*, 3rd ed. Boston: Addison-Wesley, 2000.

[3] A. Hejlsberg, S. Wiltamuth, and P. Golde, *The C# Programming Language*. Boston, Massachusetts: Addison-Wesley Professional, 2003.

[4] D. C. Schmidt, "The ADAPTIVE Communication Environment (ACE)," www.cs.wustl.edu/∼schmidt/ACE.html, 1997.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[6] *The Common Object Request Broker: Arch. and Specification*, OMG, 2002.

[7] SUN, "Java Remote Method Invocation (RMI) Specification," java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/ rmi-TOC.doc.html, 2002.

[8] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill, "TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems," *IEEE Distributed Systems Online*, vol. 3, no. 2, Feb. 2002.

[9] C. Szyperski, *Component Software — Beyond Object-Oriented Programming*. Reading, Massachusetts: Addison-Wesley, 1998.

[10] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000.

[11] Sun Microsystems, "Enterprise JavaBeans Specification," java.sun.com/products/ejb/docs.html, Aug. 2001.

[12] Microsoft Corporation, "Microsoft .NET Development," msdn.microsoft.com/net/, 2002.

[13] *CORBA Components*, OMG Document formal/2002-06-65 ed., Object Management Group, June 2002.

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns*. New York: Wiley & Sons, 1996.

[15] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*. Atlanta, GA: ACM, Oct. 1997, pp. 184–199.

[16] T. Bray, J. Paoli, and C. M. Sperberg-McQueen (Eds), "Extensible Markup Language (XML) 1.0 (2nd Edition)," W3C Recommendation, 2000. [Online]. Available: citeseer.nj.nec.com/bray00extensible.html

[17] G. Karsai, S. Neema, B. Abbott, and D. Sharp, "A Modeling Language and Its Supporting Tools for Avionics Systems," in *Proceedings of 21st Digital Avionics Systems Conf.*, Aug. 2002.

[18] D. C. Sharp and W. C. Roll, "Model-Based Integration of Reusable Component-Based Avionics System," in *Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

[19] W. Roll, "Towards Model-Based and CCM-Based Applications for Real-time Systems," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*. IEEE/IFIP, May 2003.

[20] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal of Computer Simulation, Special Issue on Simulation Software Development Component Development Strategies*, vol. 4, Apr. 1994.

[21] C. Lüer and D. S. Rosenblum, "Wren—an environment for component-based development," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 207–217, 2001.

[22] G. T. Heineman and B. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*. Reading, Massachusetts: Addison-Wesley, 2001.

[23] M. Odersky and M. Zenger, "Scalable component abstractions," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*. New York, NY, USA: ACM Press, 2005, pp. 41–57.

[24] Y. Smaragdakis and D. Batory, "Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 215–255, 2002.

[25] K. Ostermann, "Dynamically composable collaborations with delegation layers," in *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 2002, pp. 89–110.

[26] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin, "Generating product-lines of product-families," in *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2002, p. 81.

[27] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 187–197.

[28] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Reading, Massachusetts: Addison-Wesley, 2000.

[29] M. Mezini and K. Ostermann, "Variability management with feature-oriented programming and aspects," in *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*. New York, NY, USA: ACM Press, 2004, pp. 127–136.

[30] C. Prehofer, "Feature-oriented programming: A fresh look at objects," in *ECOOP'97—Object-Oriented Programming, 11th European Conference*, M. Aksit and S. Matsuoka, Eds., vol. 1241. Jyväskylä, Finland: Springer, 9–13 1997, pp. 419–443. [Online]. Available: citeseer.nj.nec.com/195556.html

[31] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997, pp. 220–242.

[32] S. R. Ponnekanti and A. Fox, "Sword: A developer toolkit for web service composition," 2002. [Online]. Available: http://www.citebase.org/cgi-bin/citations?id=oai:wwwconf.ecs.soton.ac.uk:226

[33] N. Milanovic and M. Malek, "Current solutions for web service composition," *IEEE Internet Computing*, vol. 8, no. 6, pp. 51–59, 2004.

[34] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems," in *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, May 2003.

[35] J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis, "VEST: An Aspect-based Composition Tool for Real-time Systems," in *Proceedings of the IEEE Real-time Applications Symposium*. Washington, DC: IEEE, May 2003, pp. 58–69.

[36] S. Kodase, S. Wang, Z. Gu, and K. G. Shin, "Improving Scalability of Task Allocation and Scheduling in Large Distributed Real-time Systems using Shared Buffers," in *Proceedings of the 9th Real-time/Embedded Technology and Applications Symposium (RTAS)*. Washington, DC: IEEE, May 2003.

[37] R. Chinnici, J.-J. Moreau, C. A. Ryman, and S. Weerawarana, "Web services description language (wsdl) version 2.0 part 1: Core language," W3C Candidate Recommendation, 2006. [Online]. Available: www.w3.org/TR/2006/CR-wsdl20-20060106/

[38] *Unified Modeling Language (UML) v1.4*, OMG Document formal/2001-09-67 ed., Object Management Group, Sept. 2001.

[39] Object Technology International, Inc., *Eclipse Platform Technical Overview: White Paper*, Updated for 2.1, Original publication July 2001 ed., Object Technology International, Inc., Feb. 2003.

[40] C. Lüer, "Evaluating the eclipse platform as a composition environment," in *3rd international workshop on Adoption-Centric Software Engineering ACSE 2003, ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 789–790.

[41] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "A platform-independent component modeling language for distributed real-time and embedded systems," *Elsevier Journal of Computer and System Sciences*, 2005.

[42] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, pp. 44–51, November 2001.

[43] A. Gokhale, D. C. Schmidt, B. Natarajan, and N. Wang, "Applying Model-Integrated Computing to Component Middleware and Enterprise Applications," *The Communications of the ACM Special Issue on Enterprise Components, Service and Business Rules*, vol. 45, no. 10, Oct. 2002.

[44] *Unified Modeling Language: OCL version 2.0 Final Adopted Specification*, OMG Document ptc/03-10-14 ed., Object Management Group, Oct. 2003.

[45] K. Balasubramanian, A. S. Krishna, E. Turkay, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems," *International Journal of Embedded Systems special issue on Design and Verification of Real-time Embedded Software*, Apr. 2005.

[46] K. Balasubramanian, A. Gokhale, J. Gray, Y. Lin, J. Zhang, and J. Gray, "Weaving Deployment Aspects into Domain-Specific Models," *International Journal on Software Engineering and Knowledge Engineering: Special*

Issue on Aspect-Oriented Software Design Models, 2006 (Conditionally Accepted).

[47] Software Composition and Modeling (Softcom) Laboratory, "Constraint-Specification Aspect Weaver (C-SAW)," http://www.cis.uab.edu/gray/Research/C-SAW, University of Alabama at Birmingham, Birmingham, AL.

[48] J. Gray, T. Bapty, and S. Neema, "Handling Crosscutting Constraints in Domain-Specific Modeling," *Communications of the ACM*, pp. 87–93, October 2001.

[49] C. Elliot, "Makefile, project and workspace creator (mpc)," www.ociweb.com/products/mpc, 2006.

[50] J. DeTreville, "Making System Configuration More Declarative," in *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (Hot OS X)*, Santa Fe, NM, June 2005.

[51] L. Cons and P. Poznanski, "Pan: A high-level configuration language," in *Proceedings of 16th System Administration Conference (LISA 2002)*, Berkeley, CA, 2002, pp. 83–98.

[52] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," in *Proceedings of the Usenix Annual Technical Conference (USENIX '04)*, Boston, MA, June 2004.

[53] H. S. Thompson, D. Beech, M. Maloney, and N. M. et al., "XML Schema Part 1: Structures," W3C Recommendation, 2001. [Online]. Available: www.w3.org/TR/xmlschema-1/

[54] P. V. Biron and A. M. et al., "XML Schema Part 2: Datatypes," W3C Recommendation, 2001. [Online]. Available: www.w3.org/TR/xmlschema-2/

[55] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "An infrastructure for the rapid development of xml-based architecture description languages," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM Press, 2002, pp. 266–276.

[56] M. Rose, "The Blocks Extensible Exchange Protocol (BEEP) Core," *IETF Network Working Group Request for Comments, RFC 3080*, pp. 1–58, Mar. 2001.

[57] W3C, "Simple Object Access Protocol (SOAP) 1.1," www.w3c.org/TR/SOAP, May 2000.

[58] J. Richter, *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002.

[59] *Deployment and Configuration Adopted Submission*, Document ptc/03-07-08 ed., OMG, July 2003.

[60] Krishnakumar Balasubramanian and Aniruddha Gokhale and Gabor Karsai and Janos Sztipanovits and Sandeep Neema, "Developing applications using model-driven design environments," *Computer*, vol. 39, no. 2, pp. 33–40, 2006.

[61] N. Wang, D. C. Schmidt, K. Parameswaran, and M. Kircher, "Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation," in *24th Computer Software and Applications Conference*. Taipei, Taiwan: IEEE, Oct. 2000.

[62] A. Krishna, A. Gokhale, D. C. Schmidt, J. Hatcliff, and V. Ranganath, "Context-Specific Middleware Specialization Techniques for Optimizing Software Product-line Architectures," in *Proceedings of EuroSys 2006*. Leuven, Belgium: ACM, Apr. 2006.

[63] R. Klefstad, A. S. Krishna, and D. C. Schmidt, "Design and Performance of a Modular Portable Object Adapter for Distributed, Real-time, and Embedded CORBA Applications," in *Proceedings of the 4th International Symposium on Distributed Objects and Applications*. Irvine, CA: OMG, October/November 2002.

[64] A. S. Krishna, D. C. Schmidt, R. Klefstad, and A. Corsaro, "Towards Predictable Real-time Java Object Request Brokers," in *Proceedings of the 9th Real-time/Embedded Technology and Applications Symposium (RTAS)*. Washington, DC: IEEE, May 2003.

[65] A. S. Krishna, D. C. Schmidt, and R. Klefstad, "Enhancing Real-time CORBA via Real-time Java," in *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*. Tokyo, Japan: IEEE, May 2004.

[66] Egon Teiniker and Stefan Mitterdorfer and Christian Kreiner and Zsolt Kovács and Reinhold Weiss, "Local Components and Reuse of Legacy Code in the CORBA Component Model," in *Proceedings of the* 28^{th} *Euromicro Conference (EUROMICRO'02)*. Dortmund, Germany: IEEE, September 2002, pp. 4–9.

[67] A. Diaconescu and J. Murphy, "Automating the performance management of component-based enterprise systems through the use of redundancy," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM Press, 2005, pp. 44–53.

[68] G. Singh and S. Das, "Customizing event ordering middleware for component-based systems." in *ISORC*. IEEE Computer Society, 2005, pp. 359–362.

[69] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar, "Application specific data replication for edge services," in *WWW '03: Proceedings of the 12th international conference on World Wide Web*. New York, NY, USA: ACM Press, 2003, pp. 449–460.

[70] N. K. Mukhi, R. Konuru, and F. Curbera, "Cooperative middleware specialization for service oriented architectures," in *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*. New York, NY, USA: ACM Press, 2004, pp. 206–215.

[71] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan, "Skoll: Distributed Continuous Quality Assurance," in *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*. Edinburgh, Scotland: IEEE/ACM, May 2004.

[72] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. New York: John Wiley & Sons, 2004.