International Journal of Software Engineering
and Knowledge Engineering
Vol. 16, No. 3 (2006) 403–424
© World Scientific Publishing Company

World Scientific
www.worldscientific.com

WEAVING DEPLOYMENT ASPECTS INTO DOMAIN-SPECIFIC MODELS*

KRISHNAKUMAR BALASUBRAMANIAN[†] and ANIRUDDHA GOKHALE[‡]

Department of Electrical Engineering and Computer Science,
Vanderbilt University, P.O. Box 1829, Station B, Nashville, TN 37235, USA
[†] *kitty@dre.vanderbilt.edu*
[‡] *gokhale@dre.vanderbilt.edu*

YUEHUA LIN[§], JING ZHANG[¶] and JEFF GRAY[‖]

Department of Computer and Information Sciences,
University of Alabama at Birmingham,
1300 University Boulevard, Birmingham, AL 35294, USA
[§] *liny@cis.uab.edu*
[¶] *zhangj@cis.uab.edu*
[‖] *gray@cis.uab.edu*

Domain-specific models increase the level of abstraction used to develop large-scale component-based systems. Model-driven development (MDD) approaches (e.g., Model-Integrated Computing and Model-Driven Architecture) emphasize the use of models at all stages of system development. Decomposing problems using MDD approaches may result in a separation of the artifacts in a way that impedes comprehension. For example, a single concern (such as deployment of a distributed system) may crosscut different orthogonal activities (such as component specification, interaction, packaging and planning). To keep track of all entities associated with a component, and to ensure that the constraints for the system as a whole are not violated, a purely model-driven approach imposes extra effort, thereby negating some of the benefits of MDD.

This paper provides three contributions to the study of applying aspect-oriented techniques to address the crosscutting challenges of model-driven component-based distributed systems development. First, we identify the sources of crosscutting concerns that typically arise in model-driven development of component-based systems. Second, we describe how aspect-oriented model weaving helps modularize these crosscutting concerns using model transformations. Third, we describe how we have applied model weaving using a tool called the Constraint-Specification Aspect Weaver (C-SAW) in the context of the Platform-Independent Component Modeling Language (PICML), which is a domain-specific modeling language for developing component-based systems. A case study of a joint-emergency response system is presented to express the challenges in modeling a typical distributed system. Our experience shows that model weaving is an effective and scalable technique for dealing with crosscutting aspects of component-based systems development.

Keywords: Component-based middleware; deployment and configuration; aspect modeling; domain-specific modeling.

*This work was sponsored in part by AFRL Contract# F33615-03-C-4112 for DARPA PCES Program, Raytheon, and Siemens CT.

1. Introduction

Model-driven development (MDD) is emerging as a new paradigm to develop complex component-based distributed systems. By promoting models to the status of a first-class entity in the design and implementation of such systems, developers can reason about systems at a much higher level of abstraction than by using purely programmatic techniques. Reusable approaches to distributed systems development based on component middleware technologies, such as CORBA Component Model (CCM) [1], .NET [2] and J2EE [3], have yielded a paradigm shift from focusing on building individual components to composition and integration of systems from a set of pre-built, reusable components. MDD-based approaches lend themselves well to composition and integration-related tasks since they emphasize a visual approach to system development, which is crucial to composition and integration activities. MDD also permits the description of a system using constraint languages [4], which can be enforced during design-time to prevent common errors that may otherwise occur late in the integration stage. A further benefit of MDD is that it makes the task of system analysis easier by providing better abstractions and notations that are closer to the domain of the system. As such, MDD helps to shield system developers from changes in the underlying middleware platform.

1.1. *MDD challenges*

Although MDD approaches are desirable in large-scale component-based distributed system development, the promotion of models to the status of first-class entities incurs other challenges, wherein system developers are exposed to a number of crosscutting concerns at the modeling level [5]. The crosscutting concerns include activities like composition of sub-systems from individual components, configuration of the different components, integration of systems using components from different vendors, and deployment of such composed systems. In Sec. 2, we showcase the details of these concerns as applied to component-based distributed system development.

An example of such a concern is that of keeping track of the dependencies on the run-time environment for every component in a system. Prior to MDD, this dependency was captured at the implementation level using scripts, but often the implications of the dependency were ignored in the implementation. If any modification was made at the system level (e.g., if a component was removed, or a new component was added), the scripts that manage the run-time dependencies must be updated manually. This is a tedious and error-prone task. In an MDD-based approach, such dependencies are captured at the modeling level using elements of the language. Although MDD makes the task of keeping track of the run-time dependencies easier, it still is error-prone to modify the dependencies manually in a system that has a large number of components. Addressing these crosscutting concerns using conventional MDD approaches can increase the type and number of elements that need to be manipulated at the modeling level, which may negate some of the benefits offered by MDD. What is desired is an enhanced MDD approach that is scalable (i.e., it should be easy to perform modifications to the model even in the presence of a large number of components), and gives assurance that changes to model elements keep the model in a consistent state.

1.2. *Solution approach → Aspect-oriented model weaving*

A promising approach to address the problems associated with applying MDD to large-scale distributed systems development is aspect-oriented model weaving [6], which unites the ideas of aspect-oriented software development (AOSD) [7] with MDD to provide better modularization of properties that crosscut multiple layers of a model [5].

Our approach to improving the scalability of MDD [8] for component-based distributed system development — and subsequently untangling the crosscutting concerns at the modeling level — relies on applying aspect-oriented weaving to domain-specific modeling languages (DSMLs). In this paper, we illustrate our ideas in the context of a sample distributed system scenario by applying the *Constraint-Specification Aspect Weaver* (C-SAW) [9], which is an aspect-oriented model weaver, to the *Platform-Independent Component Modeling Language* (PICML) [10].

PICML is an open-source DSML (available for download at www.dre.vanderbilt.edu/cosmic) developed using the *Generic Modeling Environment* (GME) [11]. PICML enables developers of component-based distributed systems to define component interfaces, along with their properties and system software building rules. PICML also provides generative tools to synthesize valid XML descriptor files that enable automated system deployment. C-SAW is a model transformation engine that can be used to describe the essence of a model-based crosscutting concern and transform a model accordingly. In C-SAW, aspects are defined at the modeling level using the *Embedded Constraint Language* (ECL). C-SAW assists model engineers in rapidly inserting and removing new properties and policies into models without the need for extensive manual adaptation. This paper examines the benefits that can be achieved by combining the aspect-oriented model weaving supported by C-SAW with PICML's MDD-based approach to distributed systems development. The primary combination of this synergy closes a significant gap in developing and deploying component-based distributed systems.

Paper organization. The remainder of this paper is organized as follows: Section 2 evaluates the use of MDD for distributed component systems by using an unmanned air vehicle (UAV) application as a running example; Sec. 3 gives an overview of the aspect-oriented model weaving approach, illustrates how we have applied it to the UAV example developed using PICML, and showcases the benefits of this approach; Sec. 4 compares our work with other tools that apply aspect-oriented approaches to distributed component systems development; and Sec. 5 presents concluding remarks.

2. Evaluating Model-Driven Development Approaches to Developing Component-Based Systems

MDD provides numerous benefits over programmatic approaches to large-scale component-based distributed systems development [12]. However, MDD also incurs challenges due to scalability and crosscutting concerns, similar to the challenges seen in programmatic approaches. Hence, it is imperative to enhance MDD to address these challenges.

To illustrate the crosscutting and scalability challenges in MDD for component-based distributed systems, we first present a brief overview of MDD approaches highlighting a DSML which we have developed for component-based distributed systems. We then use a component-based distributed system case study to illustrate how the crosscutting challenges manifest themselves in MDD. Our case study is an emergency response system that uses multiple unmanned air vehicles (UAVs) to perform aerial imaging, survivor tracking and damage assessment. Using the UAV scenario, we then highlight the scalability challenges and crosscutting concerns a model engineer faces when building a system like the emergency response system.

2.1. *Overview of model-driven development of component-based systems*

MDD is a paradigm that focuses on using models to describe many system development activities (i.e., models provide input and output at all stages of system development until the final system itself is generated). In MDD, models are used to describe all artifacts of the system (e.g., interfaces, interactions, and properties of all the components that comprise the system). These models can be manipulated in a number of different ways to analyze the system, and in some cases to generate the complete implementation of the system. In order to capture the semantics in an effective manner that is as close as possible to the domain of the developed system, a DSML can be used. A DSML is a five-tuple [13] consisting of:

- **Concrete syntax (C)**, which defines the notation used to express domain entities,
- **Abstract syntax (A)**, which defines the concepts, relationships and integrity constraints available in the language,
- **Semantic Domain (S)**, which defines the formalism used to map the semantics of the models to a particular domain,
- **Syntactic mapping ($\mathbf{M}_C$: A $\rightarrow$ C)**, which assigns syntactic constructs (e.g., graphical and/or textual) to elements of the abstract syntax,
- **Semantic mapping ($\mathbf{M}_S$: A $\rightarrow$ S)**, which relates the syntactic concepts to those of the semantic domain.

Crucial to the success of DSMLs is *metamodeling* and *auto-generation.* A *metamodel* defines the elements of a DSML that are tailored to a particular domain, such as the domain of avionics mission computing or emergency response systems.

Auto-generation involves automatically synthesizing artifacts from models, thereby relieving DSML users from the accidental complexities of the artifacts themselves, including their format, syntax, or semantics. Examples of such artifacts include (but are not limited to) code in some programming language, and descriptors (in formats such as XML) that can serve as input to other tools.

To support effective design and development of component-based systems, we have defined the *Platform-Independent Component Modeling Language* (PICML) [10, 14, 15] using the *Generic Modeling Environment* (GME) [11]. GME is a meta-programmable modeling environment with a general-purpose editing engine, separate view-controller GUI, and a configurable persistence engine. Because GME is meta-programmable, the same environment used to define PICML is also used to build models, which are instances of the PICML metamodel.

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 the different types of modeling elements that are essential for developing, composing, configuring and deploying component-based systems. The artifacts pertaining to configuration and deployment of component-based systems that are generated from PICML are then deployed using the Component-Integrated ACE ORB (CIAO) [16, 17], which is an implementation of the CCM.[a]

In terms of the five-tuple defined above, PICML uses bitmap-based icons representing elements of the platform (e.g., CCM) as the concrete syntax. For elements like components, PICML also uses enhanced visualizations called "decorators," which display the different ports of a component and allow connections to be made between ports of different components. The abstract syntax of PICML is defined using a variant of UML class diagrams available in GME. In addition to the class diagrams, OMG's Object Constraint Language (OCL) is used to enforce the semantics that are not captured by the class diagrams. The semantic domain of PICML is the CCM platform. Thus, the semantics of the different elements are governed by the CCM specification, which defines the elements as well as valid interactions between the different elements that make up a system built using CCM. The syntactic mapping between the different elements in the abstract syntax to the elements in the concrete syntax is achieved by associating each element in the metamodel with icons, or with a decorator (in the case of special elements like components and assemblies). The semantic mapping associating the elements in the metamodel with elements in the CCM platform is performed partly using the OCL constraints, and partly using the various model interpreters defined in PICML. A *model interpreter* is a GME plug-in written using a high-level language like C++, and can be used to enforce the semantics not captured by OCL constraints alone. Interpreters for PICML are also used to generate various artifacts like component

[a]PICML is currently being enhanced to model systems using other standards-based component middleware, such as J2EE and .NET, in addition to CCM.

configuration files and deployment plans, which are needed for the deployment of CCM components.

The PICML metamodel defines approximately 115 different types of basic elements, with 57 different types of associations between these elements grouped under 14 different folders. The PICML metamodel also defines 222 constraints that are enforced by GME's constraint manager during the design process. Using GME tools, the PICML metamodel can be compiled into a *modeling paradigm*, which defines a domain-specific modeling environment. From this metamodel, approximately 20,000 lines of C++ code (used to represent 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. 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. PICML currently has 8 interpreters using 222 generated C++ classes and approximately 8,000 lines of handwritten C++ code to generate the following descriptors needed to support the OMG Deployment and Configuration (D&C) specification [18]:

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

2.2. *A representative case study using PICML*

This section presents an emergency response system as the guiding example to illustrate the MDD approach, and to illustrate the challenges that arise in modeling distributed component systems. This example models emergency response situations (such as disaster recovery efforts stemming from floods, earthquakes, or hurricanes) and consists of a number of interacting subsystems. Our focus in this paper is on the composition, integration and deployment of a UAV, which is used to monitor terrain for flood damage, spot survivors that need to be rescued, and

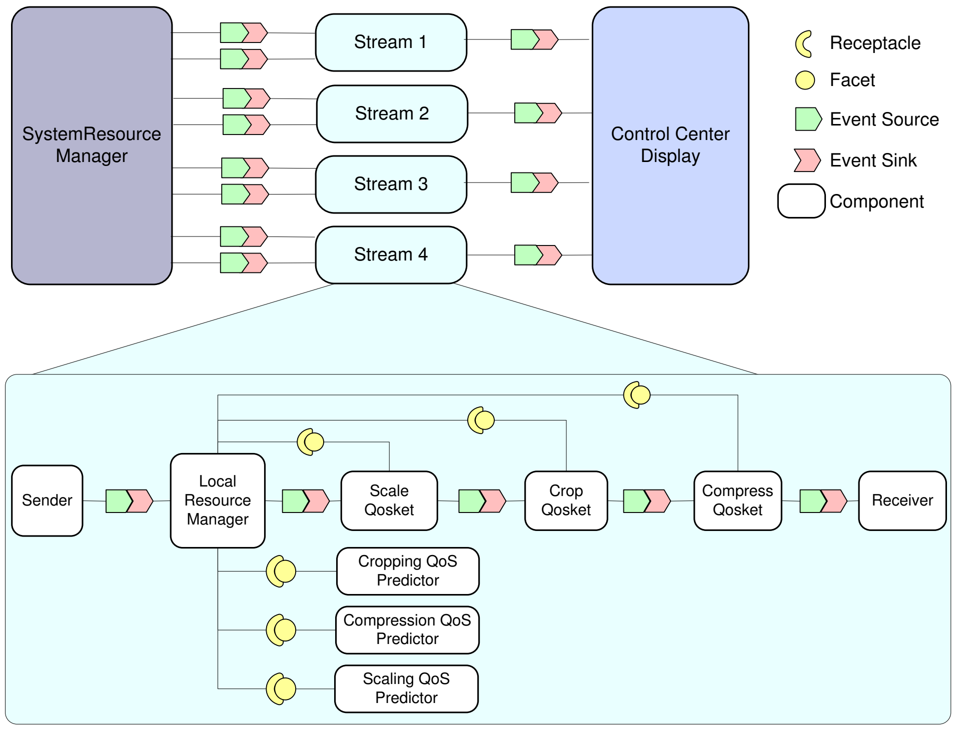


Fig. 1. Emergency response system components.

assess the extent of damage. The UAV transmits this imagery to various other emergency response units. The software components of this UAV application are shown in Fig. 1 and described in detail in [10].

The UAV application involves sending streams of images from each UAV to a control center responsible for monitoring the image data. Each image stream is composed of a `Sender` (e.g., the UAV), a number of `Qosket` components, and a `Receiver` component. `Sender` components are responsible for collecting the images from each image sensor on the UAV. The `Sender` passes the images to a series of `Qosket` [16] components. Qoskets are software components that encapsulate a set of contracts, a set of system condition objects and performance adaptation behavior logic. Thus, Qoskets perform adaptations on the images to ensure that the images can be transmitted without violating the quality of service (QoS) requirements. Examples of Qosket components include `CompressQosket`, `ScaleQosket`, `CropQosket`, `PaceQosket`, and a `DiffServQosket`. The final `Qosket` in the pipeline then passes the images to a `Receiver` component, which collects the images and passes them on to a display in the control room of the emergency response team.

2.3. *Scalability and crosscutting challenges in applying MDD to component-based distributed systems*

This section describes how DSMLs (e.g., PICML) applied to component-based distributed systems (e.g., a UAV) incur different scalability challenges and crosscutting concerns. We illustrate these challenges as they are manifested in each of the modeling stages of systems development described below. Although the description illustrates these challenges as they emerge in PICML, we believe similar challenges will exist in any DSML used to develop large-scale component-based distributed systems.

2.3.1. *Crosscutting concerns in modeling interface definitions*

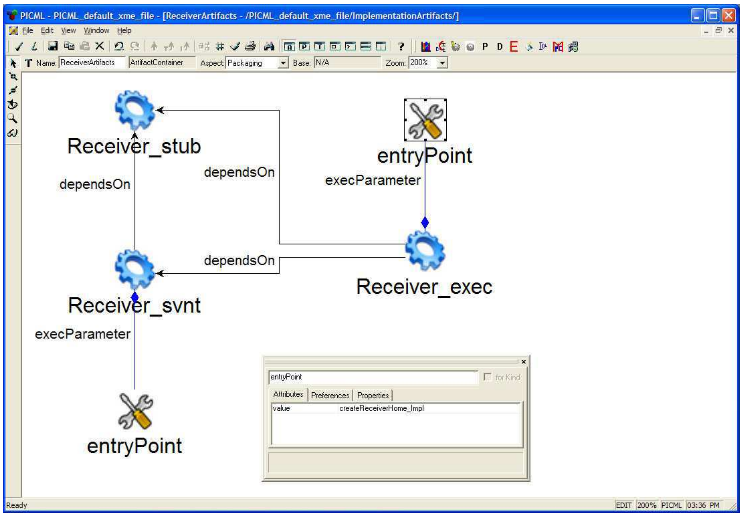
PICML assists in modeling the individual component types of a system, which involves either importing the component interface definitions from existing interface definition language (IDL) files, or explicitly modeling them using PICML. In our example, this involves defining the interfaces for the `Sender`, `Receiver`, `Qosket`, `SystemResourceManager`, and the `LocalResourceManager` components.

In order to deploy a system using component middleware, such as CIAO, the individual components that together realize the application must be specified. This step is very crucial because the type (indicated by its name, such as `LocalResourceManagerComponent`) of the individual monolithic components is defined at this stage. The interface of the system with external entities is also defined during this stage.

These definitions (including the names) serve as a bridge between the entities defined at the modeling level and the corresponding implementation. Such component definitions are scattered throughout the system model through the use of references to the individual component types. For example, the component instances that are used to define the component interactions are instances of the individual component types. Thus, it is the model engineer's responsibility to maintain the one-to-many relationship between the component types and the different instances of the same type that are scattered across the models. If a component type is modified/deleted, then the model engineer has to manually update/remove all the references scattered in the remaining model. This is an inherently time-consuming and error-prone task when performed manually, and does not scale when the number of components in the model increase. For example, addition of new ports to a component type that is part of a stream in the UAV scenario results in a need to modify all the existing uses of that component type throughout the model.

2.3.2. *Modeling implementation artifact definitions*

During this stage, a model engineer defines the implementation artifacts shown in Fig. 2 for each monolithic component, which involves defining the different implementation artifacts (e.g., shared libraries) that each component depends on, as well



Name: ReceiverArtifacts ArtifactContainer Aspect: Packaging Base: N/A Zoom: 200%

entryPoint

value createReceiverHome_Impl

Fig. 2. Implementation artifact definition.

as describing the dependencies that each component may have on external system libraries.

For example, when building the UAV application using CIAO, a monolithic component, such as `SystemResourceManager`, is composed of three libraries, (1) `SystemResourceManager_exec`, which contains the implementation of the component functionality, (2) `SystemResourceManager_stub`, which contains code that provides the marshaling and de-marshaling related functionality for each component, and (3) `SystemResourceManager_svnt`, which contains the code to glue together the component with other portions of the execution environment, such as the underlying CORBA middleware infrastructure. Any error in capturing the dependencies will result in run-time failure of the components. Although the number, names and kinds of implementation artifacts might differ across different middleware implementations, each component in a distributed system will end up having dependencies on the artifacts that are necessary to provide the functionality of the component. Thus, these artifacts need to be modeled explicitly, an activity that does not scale well as the number of components and their corresponding implementation artifacts increase. Moreover, the model engineer is also responsible for maintaining the dependencies as the system evolves. Any error in the maintenance of the dependency will also result in a run-time error due to unresolved dependencies on implementation artifacts of component instances.

Another example of a concern associated with an implementation is the need to follow specific naming conventions as imposed by the underlying middleware, wherein the naming of implementation artifacts in the model must mirror the naming conventions of the underlying component middleware infrastructure.

For example, in the default configuration of CIAO, if the three dependent libraries for `SystemResourceManager` are not named `SystemResourceManager_exec`, `SystemResourceManager_stub`, and `SystemResourceManager_svnt` respectively, it will result in a run-time error. 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.

Yet another naming related crosscutting concern is with the specification of the entry point for loading a component implemented as a shared library in a language like C++. Because the C++ compiler mangles the names of the methods in classes, method names that need to be exported are marked using special `extern "C"` tags. Only names that are marked using these tags are available for invocation by clients that dynamically load these components. In order to ensure that a component can be dynamically loaded, the model engineer must make sure that the definitions of the entry points defined in the implementation artifacts of a component actually map to entry points defined in the shared libraries.

2.3.3. *Modeling interaction definitions*

During this stage of development, a model engineer defines the different interactions between components, which involves composing the application from a set of individual components. The components are connected using their ports to form assemblies, which could be nested. In PICML, assemblies contain monolithic components that are connected together. Assemblies can also be hierarchical (i.e., an assembly can contain other assembly components). In our example, each stream of images is modeled as an assembly by connecting the `Sender`, `LocalResourceManager`, `Qoskets`, and the `Receiver`, as shown in Fig. 3. This assembly is then instantiated multiple times depending on the number of UAVs, along with the `SystemResourceManager`, and the `ControlCenterDisplay`, to form the complete UAV application.

There are several cases when much effort is required to manually add (or remove) new links between components:

1. If a component is removed from an application,
2. If a new port is to be connected to all external uses of a component,

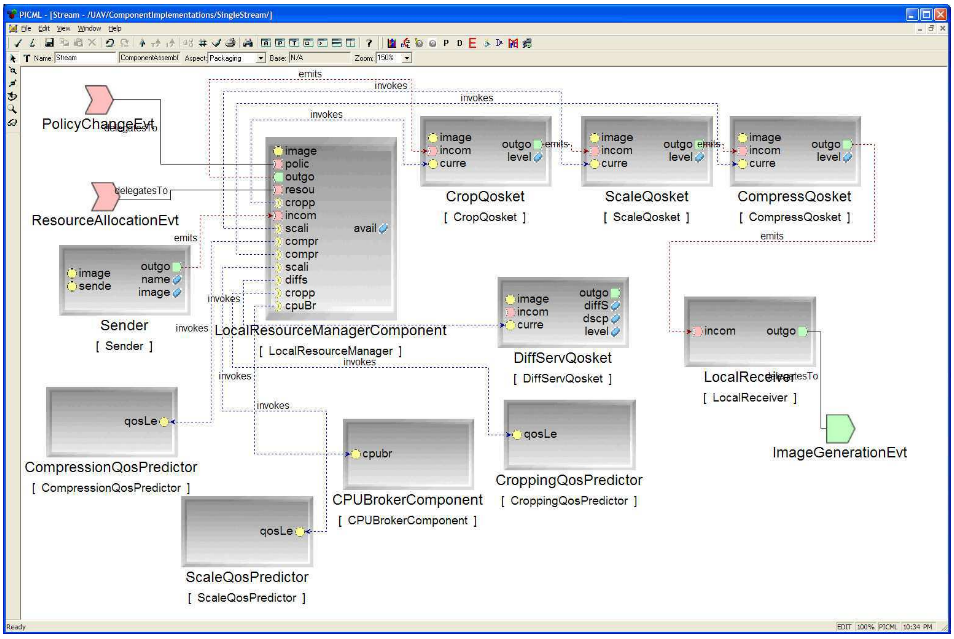


Fig. 3. Interaction definition.

3. Or, if a component is replaced by another component.

In cases where the number of components is large, the amount of effort required to perform these modifications manually is equivalent to starting the modeling effort from scratch. In the UAV example, the addition of a new management interface to all the `Qosket` components will result in extensive changes to every assembly that models a stream. The amount of effort grows exponentially because a single assembly is instantiated multiple times, depending upon the number of UAVs that are desired. Even if the required changes are performed manually, it is necessary to ensure that the changes are correct by checking the constraints in the model. In case there are violations, the correction of the constraint violation also needs to be done manually. Thus, changes to the interactions between components necessitate similar changes at multiple locations — scattered across the whole model — in a repetitive fashion.

2.3.4. *Modeling package definitions*

The packaging of component assemblies involves defining the relation between units of deployment called *packages*, and the individual assemblies that are the output of the composition process defined earlier. A package is associated with a top-level assembly, and is used to bootstrap the deployment of the application. For example, in the UAV application, the top-level assembly that contains many individual streams as sub-assemblies needs to be associated with a package so that the UAV application can be deployed. Packages are the units of deployment in CCM. It is also true for other middleware technologies like EJB (e.g., EJB uses an .ear file to initiate deployment). In order for a deployment tool to be successful in deploying an application, each package needs to be associated with metadata that describes the capabilities of each package. Examples of such metadata include the interfaces that are implemented by the component(s) that are contained in a package, dependencies on the run-time environment by components in a package, and any startup ordering dependencies on external components.

Thus, it is necessary to maintain the consistency of a package during system evolution. For example, any change in the interface definition needs to result in an appropriate change in the packages containing components that implement that interface. Any change in the dependent implementation artifacts of a component implementation also requires that the corresponding component package be updated with these artifacts. For example, in the UAV application, any change to the composition of a stream by addition or removal of a `Qosket` component necessitates appropriate changes to be made to the component packages. The need to ensure that changes to some elements are propagated to a dependent element indicates the emergence of another crosscutting concern.

2.3.5. *Modeling the domain definition and component mapping*

Domain definition involves modeling the elements of the target domain and a mapping between component instances and the target domain. This task is usually done by a domain administrator who has knowledge of the physical infrastructure on which the application is to be deployed. After the elements of the target domain are defined, a mapping between the individual component instances and assemblies onto elements of the target domain is specified. This activity results in the creation of a deployment plan, which is used by the run-time infrastructure to deploy the application.

In order to ensure successful deployment of the application, the mapping between the component instances (or assemblies) and nodes of the target domain needs to be consistent. Although constraints can help in matching the capabilities of each node with the requirements of the individual components, all the components (or assemblies) need to be assigned to nodes, and this assignment needs to be updated when the definitions of the component assemblies change. Any error in this process shows up only at run-time, which is very late in the lifecycle and proves to be very expensive to correct. For example, in the UAV example, if a new type of Qosket (e.g., `DiffServQosket`, which assists in enforcing network priorities) is added to each stream, it is necessary to ensure that the `DiffServQosket` is mapped to a UAV. Without the `DiffServQosket`, a UAV will not be able to perform QoS adaptations based on network priorities, and images that are critical to a mission might show up in the control center after images that are peripheral. This would potentially undermine the importance of the images sent by the UAV.

The use of MDD to develop systems like the UAV application provides a significant improvement over programmatic approaches based on using only component middleware. However, as outlined in the UAV context, a number of tangled concerns and scalability issues emerge in the modeling of component-based distributed systems. The concept of a component pervades these artifacts, and the challenges that occur are due to the tangling of the concerns associated with a component at multiple places in the model. If left unresolved, these challenges can hamper developer productivity, and also negatively affect the correctness of the system being modeled. Section 3 describes our solution to these problems.

3. Applying Aspect-Oriented Model Weaving to PICML

This section presents a solution to the crosscutting and scalability challenges of modeling large-scale distributed systems described in Sec. 2.3. Our approach to resolving these challenges relies on the use of aspect-oriented model weaving using C-SAW. We first provide an overview of aspect-oriented modeling and then describe our solution.

3.1. *Overview of aspect-oriented modeling*

A distinguishing feature of AOSD is the notion of crosscutting, which characterizes the phenomenon whereby some representation of a concern is scattered among multiple boundaries of modularity, and tangled amongst numerous other concerns. Aspect-Oriented Programming (AOP) languages, such as AspectJ [19], permit the separation of crosscutting concerns into aspects.

We have found that the same crosscutting problems that arise in code also exist in domain-specific models [5]. For example, it is often the case that the metamodel forces a specific type of decomposition, such that the same concern is repeatedly applied in many places, usually with slight variations at different nodes in the model. This is a consequence of the "dominant decomposition" [20], which occurs when a primary modularization strategy is selected that subjects other concerns to be described in a non-localized manner.

Aspect-Oriented Modeling (AOM) is an AOSD extension applied to earlier stages of the lifecycle. Our specific perspective of AOM improves the modeling task itself by providing the ability to specify properties across a model during the system modeling process. This action is performed by using a weaver that has been constructed for the GME modeling tool. We consider AOM to be much more than mere notations that provide traceability to latter stages of development — a model weaver can assist in the automation of the modeling process.

3.2. *Aspect modeling with C-SAW*

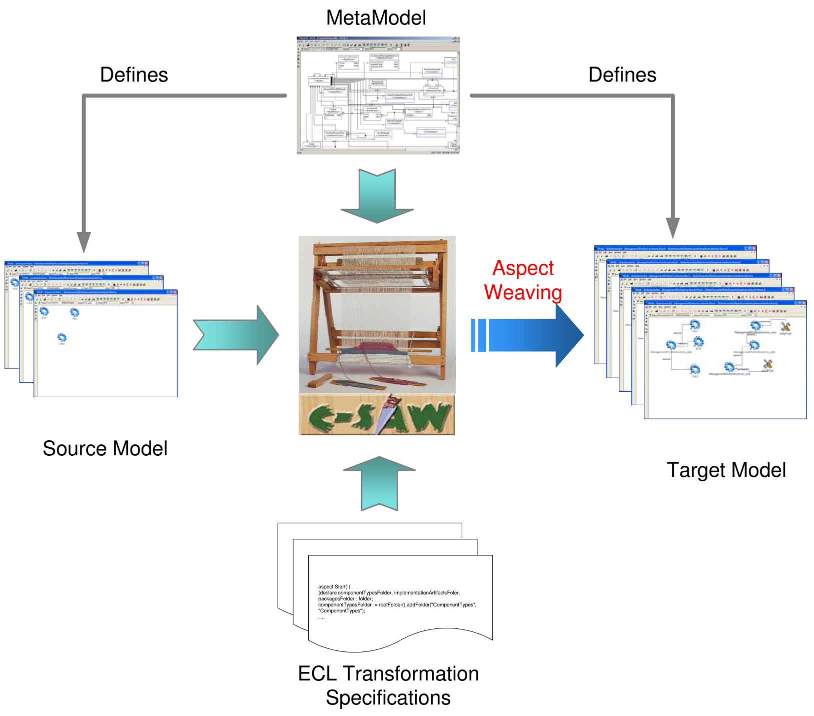
We have designed C-SAW to provide support for modularizing crosscutting modeling concerns in the GME. This weaver operates on the internal representation of a model (similar to an abstract syntax tree of a compiler). GME provides a framework that allows DSML developers to register custom actions and hooks with the environment. These hooks can read and write the elements of a model during the modeling stage. GME also provides an introspection API, which provides knowledge about the types and instances of a model, without *a priori* knowledge about the underlying DSML. Utilizing this feature of GME, we have implemented C-SAW as a "plug-in," which is GME terminology for a DSML independent hook. Thus, the benefits of C-SAW are applicable across a whole spectrum of DSMLs.

To be effective, C-SAW also requires the features of an enhanced model transformation language. Standard OCL is strictly a declarative language for specifying assertions and properties of UML models. Our need to extend OCL is motivated by the fact that we require an imperative language for describing the actual model transformations. We designed a language called the Embedded Constraint Language (ECL) to describe model transformations. ECL is an extension of the OCL and provides many of the common features of OCL, such as arithmetic operators, logical operators, and numerous operators on collections (e.g., size, and select). A unique feature of ECL that is not provided within OCL, however, is a set of reflective operators for querying models. For example, aggregation operators (e.g., `models(expression)` and `atoms(expression)`) are used to select all the models from a model container (and all the atoms from a model) that satisfy the constraint specified by the expression. These operators, together with the `select` operator, can be applied to first class model objects (e.g., a container model or primitive model element) to obtain reflective information needed to perform model weaving. In addition, ECL provides a set of transformation operators to change the state of a model such as `addModel, addAtom, removeModel, removeAtom` and `setAttribute`. The operators can change the structure and properties of a model.

The AOM approach that we have adopted in C-SAW can be summarized by the diagram in Fig. 4. As shown in this figure, model transformations are performed between the source models and the target models that belong to the same metamodel. C-SAW weaves additive changes into these source models to generate the target models relying on transformation specifications written in ECL.

- **Modeling Aspects:** A *modeling aspect* is a modular construct that specifies a crosscutting concern in a model hierarchy. A modeling aspect describes the context of a strategy call (see the definition of strategy below), which can be a specific model, atom, or connection. Like a pointcut designator in AspectJ [19], a modeling aspect is responsible for identifying a set of model nodes across a model hierarchy in a modular way, and offers the capability to make quantifiable statements across the boundaries of a model.
- **Strategies:** A *strategy* is used to specify transformation logic (e.g., constraint propagation, and the application of specific properties) to the model nodes.[b] Each

[b] "Model nodes" refer to modeling elements that are defined in the metamodel, and serve as



```
aspect Start( )
{declare componentTypesFolder, implementationArtifactsFoler,
packagesFolder : folder;
componentTypesFolder := rootFolder().addFolder("ComponentTypes",
"ComponentTypes");
....
```

Fig. 4. C-SAW aspect model weaver framework.

metamodel will have unique strategies that can be applied to a model through C-SAW. A strategy provides a hook that the weaver can call in order to process node-specific application and propagation. Thus, strategies offer numerous ways for instrumenting nodes in a model with crosscutting concerns. A strategy call in an aspect implements the binding and parameterization of the strategy to all the model nodes specified by the aspect.

3.3. *Resolving UAV crosscutting modeling challenges with C-SAW*

As described in Sec. 2.3, the modeling concern related to application deployment has been decomposed into multiple views along the dimension of the underlying CCM run-time. However, this modularization results in related concepts from the dimensions of individual components and assemblies to be non-localized and split across multiple entities. This section describes how C-SAW is used to modularize the concepts related to individual components and assemblies. The approach takes advantage of aspect-oriented model weaving to fill in the information into the various artifacts that are necessary to deploy the UAV application.

visualization elements in the domain model.

```
 1:  aspect Deploy()
 2:  {
 3:   // Create a folder called "ImplArtifacts"
 4:   implementationArtifactsFolder
 5:     := rootFolder().addFolder("ImplementationArtifacts", "ImplArtifacts");
 6:
 7:   // Create a folder called "Packages"
 8:   packagesFolder := rootFolder().addFolder("ComponentPackages", "Packages");
 9:
10:   // Retrieve component assembly, and invoke strategy
11:   rootFolder().findFolder("ComponentImplementations").models()
12:     ->select(f | f.kindOf() == "ComponentImplementationContainer")
13:       ->models()->select(p | p.kindOf() == "ComponentAssembly")
14:         ->models()->select(c | c.kindOf() == "Component")
15:           ->WeaveDeploymentArtifacts();
16: }
```

Aspect Listing 1: Deployment Modeling Aspect

The task of modularizing the concerns of deployment begins with defining a modeling aspect in C-SAW. Aspect Listing 1 shows a snippet of the definition of the `Deploy` aspect. This modeling aspect defines the tasks that a model engineer typically performs manually. It enables creation of different folders, which will contain the different orthogonal entities (e.g., implementation artifacts and component packages) needed to deploy an application using CCM. The initial creation of these folders is similar to inter-type declarations in AspectJ [19]. Specifically, lines 4 and 5 create a new `ImplementationArtifacts` folder called `ImplArtifacts` (note that the first parameter of the `addFolder()` function indicates the folder type and the second parameter represents the name of the folder) under the root folder of a UAV model. Similarly, line 8 creates a new `ComponentPackages` folder named `Packages`. This aspect has been extended to cover all the different activities that were discussed in Sec. 2.3. Due to space constraints, we have not shown all of the different entities that are created by this aspect.

After creating the required folders, the `Deploy` aspect determines all component assemblies, which contain the definitions of the component interactions. The component assemblies are discovered by the weave-time introspection facilities that are provided by ECL. As shown from line 11, a component assembly can be retrieved from the top level `ComponentImplementations` folder's `ComponentImplementationContainer` models. For each component model in the component assembly, the `WeaveDeploymentArtifacts` strategy is applied, as shown in Strategy Listing 1. Thus, this strategy is applied across model boundaries.

`WeaveDeploymentArtifacts` aggregates the different strategies that need to be applied to each individual component. For brevity, we illustrate just two such strategies — `ImplementationArtifacts` (as shown in line 3) and `PackageDefinition` (line 4) — which are necessary to solve the challenges described in Secs. 2.3.2 and 2.3.4. Several other deployment strategies have been created, but are not shown here in order to keep the example concise.

```
strategy WeaveDeploymentArtifacts()
{
  ImplementationArtifacts();
  PackageDefinition();
}
```

Strategy Listing 1: Weave Deployment Artifacts Strategy

```
 1: strategy ImplementationArtifacts()
 2: {
 3:   component := self;
 4:   componentName := component.getName();
 5:
 6:   // Create an instance of model ArtifactContainer
 7:   artContainer := rootFolder().findFolder("ImplArtifacts")
 8:     .addModel("ArtifactContainer", componentName);
 9:
10:   // Create Foo_exec, Foo_stub and Foo_svnt
11:   ia_exec := artContainer.addAtom("ImplementationArtifact", componentName + "_exec");
12:   ia_stub := artContainer.addAtom("ImplementationArtifact", componentName + "_stub");
13:   ia_svnt := artContainer.addAtom("ImplementationArtifact", componentName + "_svnt");
14:
15:   // Set the attribute "location" of Foo_exec, Foo_stub and Foo_svnt
16:   ia_exec.setAttribute("location", componentName + "_exec");
17:   ia_stub.setAttribute("location", componentName + "_stub");
18:   ia_svnt.setAttribute("location", componentName + "_svnt");
19:
20:   // Create a reference to Foo_stub
21:   ia_stubRef
22:     := artContainer.addReference("ImplementationArtifactReference", ia_stub);
23:
24:   // Create a connection between Foo_svnt and Foo_stub reference
25:   artContainer.addConnection("ArtifactDependsOn", ia_svnt, ia_stubRef);
26:
27:   // Create a connection between Foo_exe and Foo_stub reference
28:   artContainer.addConnection("ArtifactDependsOn", ia_exec, ia_stubRef);
29: }
```

Strategy Listing 2: Implementation Artifact Strategy

The `ImplementationArtifacts` strategy shown in Strategy Listing 2 is responsible for creating the different auxiliary shared libraries that are needed to implement a single monolithic component. Specifically, the strategy first retrieves the component (line 3) and its name (line 4). It then adds an ArtifactContainer model with the component name under the `ImplementationArtifacts` folder that was created in the `Deploy` aspect. From line 11 to line 18, three different component libraries are built within the ArtifactContainer model as `ImplementationArtifact` atoms by extending the component name with different suffixes (i.e., `_exec`, `_stub` and `_svnt`). Lines 21 and 22 create a reference to `_stub`. Finally, two dependencies are established in line 25 and line 28 as connections between the library atoms. It can be seen that this strategy modularizes:

- Creation of all implementation artifacts mandated by the underlying run-time,
- Creation of implementation artifacts that adhere to a specific naming convention,
- Keeping track of dependencies between a single monolithic component and its associated implementation artifacts,

- Setting attribute values (e.g., location and entry points) in shared libraries.

By modularizing the different activities associated with defining implementation artifacts and allowing for customizability based on idiosyncrasies of specific run-time environments, C-SAW helps resolve the challenge described in Sec. 2.3.2 by modularizing artifact definitions for all available components. This is a very time-consuming and error-prone task if performed manually across multiple components.

```
 1:  strategy PackageDefinition()
 2:  {
 3:    component := self;
 4:    componentName := component.getName();
 5:
 6:    // Create an instance of model PackageContainer
 7:    pkgContainer := rootFolder().findFolder("Packages")
 8:      .addModel("PackageContainer", componentName);
 9:
10:    // Create an instance of atom ComponentPackage
11:    compPackage := pkgContainer.addAtom("ComponentPackage", componentName);
12:
13:    // Create a reference to the current component
14:    componentRef := pkgContainer.addReference("ComponentRef", component);
15:
16:    // Create a connection between ComponentPackage and component reference
17:    pkgContainer.addConnection("PackageInterface", compPackage, componentRef);
18: }
```

Strategy Listing 3: Package Definition Strategy

The `PackageDefinition` strategy shown in Strategy 3 is responsible for creating a package and associating a component assembly with the component package. Line 11 creates a component package as an atom within the package container that was constructed in line 7. Line 17 builds an association connection between this component package and the component reference that was created in line 14. Similar strategies were defined to solve the challenges outlined in Secs. 2.3.1, 2.3.3 and 2.3.5. By combining specification aspects and strategies, C-SAW enhances the utility of a DSML like PICML, and resolves the challenges associated with a pure MDD-based approach to improve development of component-based distributed systems.

4. Related Work

The object-oriented and structured paradigms each had their genesis at the implementation level, and were applied afterward to earlier phases of the software lifecycle. A similar trend has occurred in the investigation of aspect-oriented concepts. Although the initial emphasis has been on aspect-oriented *programming*, there is a growing body of research that is focused on applying aspects and other new separation of concerns ideas to non-code artifacts [21]. Over the past decade, AOSD techniques have been investigated at all levels of the development lifecycle [22], including requirements engineering and early design [23], and detailed analysis and design [24].

Several researchers have investigated the application of AOSD concepts within the context of the UML [25–27]. In fact, a very successful workshop on aspect-oriented modeling (now in its 8th edition) has served as a common venue for researchers working in the new aspect modeling area [28]. These efforts have yielded guidelines for describing crosscutting concerns at higher levels of abstraction. A focal point of these efforts is the development of notational conventions that assist in the documentation of concerns that crosscut a design. Such notational conventions improve the ability to modularize a design that is described as a UML model. Moreover, the initial aspect modeling contributions have the important trait of improving the traceability of crosscutting concerns from design to implementation. From this perspective, the general goals of aspect modeling are similar to the objectives presented in this paper.

Many of the UML aspect modeling efforts have done much to improve the applicability of AOSD at the modeling level, but they generally tend to treat the concept of aspect-oriented design as an *adjective*; i.e., the focus has been on the notational and decorative attributes concerned with aspects and their representation within UML. The majority of aspect modeling research differs from the approach presented in this paper. The application of C-SAW is much more than a notational abstraction. The research into C-SAW has concentrated on the idea of building actual weavers for domain models; i.e., aspect modeling is approached as a specific application of model transformation, where models are evolved to specify crosscutting properties. The contribution of C-SAW, and the application to component-based distributed systems, is to consider aspect modeling as a *verb*. That is, viewing AOSD as a mechanism to improve the modeling task, itself, by providing the ability to quantify properties across a model during the system modeling process. This action is performed by utilizing a weaver that has been constructed with the concepts of modeling in mind. A research effort that also appears to have this goal in mind can be found in [29], which is focused on UML models, but provides transformation capabilities in addition to notational abstractions to represent aspects.

5. Concluding Remarks

Although MDD approaches to building distributed systems have inherent advantages over a purely programmatic approach, additional tools are needed to assist in modularizing crosscutting concerns that are not effectively captured by modeling languages like PICML. To address this problem, this paper described the aspect-oriented model weaving capabilities of the Constraint-Specification Aspect Weaver (C-SAW). Using the C-SAW concepts — *strategies* and *modeling aspects* — many of the problems associated with scattered pieces of deployment related artifacts and model scalability can be effectively addressed. Weaving at the modeling level is a form of transformation that enables a developer to evolve and maintain consistency across numerous views that are available in a modeling language. The key contribution is an ability to make changes across a model in many locations in an automated manner.

In the specific context of distributed component-based systems, the primary lessons learned concern the productivity benefits that can be achieved from an automated approach to evolving domain-specific models in the presence of crosscutting concerns. A manual approach to the specific challenges presented in Sec. 2.3 is tedious and error-prone. When the number of components increases, the number of steps required to make such modifications increase accordingly. Because of the crosscutting nature of model properties and constraints, the manual modification of models hampers productivity because of all the mouse clicking and typing involved in each change. An aspect model weaver like C-SAW offers needed support to evolve crosscutting concerns that appear in modeling interface and artifact definitions. Other concerns (e.g., component interactions, component packaging, and component mapping) also require the model engineer to explore various design alternatives in a way that is not practical through manual modification. The lessons learned in this context add to the growing number of application areas to which we have applied C-SAW, including mission computing avionics [6] and hardware configuration of physics-based applications [8].

There are several limitations to C-SAW that are being addressed as future work. One limitation, as shown in Fig. 4, is that C-SAW only works within a single metamodel. That is, model transformations cannot occur between models that are defined by different metamodels. This is not a problem for the types of crosscutting deployment concerns mentioned in this paper, but this limitation does prohibit more advanced transformations to be performed in other domains. We are working toward a future version of C-SAW that allows the specification of model transformations across multiple metamodels. A second limitation of C-SAW is the lack of support to help ensure the correctness of a modeling aspect and strategy. Currently, there is no capability provided to the model engineer to help assist in determining if an error exists in a C-SAW transformation. To address this issue, we are currently developing a testing and debugging capability within C-SAW. This would allow test cases to be constructed to determine if the ECL is written correctly to perform a desired transformation. If an error in the ECL exists, a debugger for the ECL would allow a model engineer to step through each line of ECL and examine the affect that occurs within the model.

In summary, this paper has demonstrated the application of aspect modeling to address the deployment concerns of distributed component-based systems modeled using PICML. The combination of MDD tools like PICML, and aspect-oriented model weavers like C-SAW, are crucial to realizing the goal of automated design and development of component-based middleware systems.

References

1. Object Management Group, *CORBA Components*, OMG Document formal/2002-06-65 edition, June 2002.
2. Microsoft Corporation, Microsoft .NET Development, msdn.microsoft.com/net/, 2002.
3. Sun Microsystems, Java^{TM} 2 Platform Enterprise Edition, java.sun.com/j2ee/index.html, 2001.
4. Object Management Group, *Unified Modeling Language: OCL version 2.0 Final Adopted Specification*, OMG Document ptc/03-10-14 edition, Oct. 2003.
5. J. Gray, T. Bapty, and S. Neema, Handling crosscutting constraints in domain-specific modeling, *Communications of the ACM*, October 2001, pp. 87–93.
6. J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale, and B. Natarajan, An approach for supporting aspect-oriented domain modeling, in *Proc. 2nd Int. Conf. on Generative Programming and Component Engineering* (*GPCE'03*), Erfurt, Germany, September 2003, pp. 151–168.
7. R. Filman, T. Elrad, M. Aksit, and S. Clarke, *Aspect-Oriented Software Development*, Addison-Wesley, Reading, MA, 2004.
8. J. Gray, Y. Lin, J. Zhang, S. Nordstrom, A. Gokhale, S. Neema, and S. Gokhale, Replicators: Transformations to address model scalability, in Lecture Notes in Computer Science, *Proc. 8th Int. Conf. Model Driven Engineering Languages and Systems* (MoDELS 2005), Montego Bay, Jamaica, November 2005, Springer Verlag, pp. 295–308.
9. 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.
10. 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, in *Proc. 11th Real-Time Technology and Application Symposium* (*RTAS '05*), San Francisco, CA, March 2005, IEEE, pp. 190–199.
11. A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, Composing domain-specific design environments, *IEEE Computer*, November 2001, pp. 44–51.
12. A. Gokhale, D. C. Schmidt, B. Natarajan, J. Gray, and N. Wang, Model driven middleware, in *Middleware for Communications*, ed. Q. Mahmoud, Wiley and Sons, New York, 2004, pp. 163–187.
13. G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, Model-integrated development of embedded software, *Proc. IEEE* **91**(1) (2003) 145–164.
14. A. S. Krishna, A. Gokhale, D. C. Schmidt, V. P. Ranganath, J. Hatcliff, and D. C. Schmidt, Model-driven middleware specialization techniques for software product-line architectures in distributed real-time and embedded systems, in *Proc. MODELS 2005 Workshop on MDD for Software Product-Lines*, Half Moon Bay, Jamaica, October 2005.
15. A. S. Krishna, A. Gokhale, D. C. Schmidt, V. P. Ranganath, and J. Hatcliff, Towards highly optimized real-time middleware for software product-line architectures, in *Proc. Work-in-Progress Session in the 26th IEEE Real-Time Systems Symposium* (*RTSS*) *2005*, Miami, USA, December 2005.
16. N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. D. Gill, QoS-enabled middleware, in *Middleware for Communications*, ed. Q. Mahmoud, Wiley and Sons, New York, 2003, pp. 131–162.
17. N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian, Configuring real-time aspects

in component middleware, in Lecture Notes in Computer Science, *Proc. Int. Symp. on Distributed Objects and Applications* (*DOA'04*), Agia Napa, Cyprus, October 2004, Vol. 3291, Springer-Verlag, pp. 1520–1537.

18. Object Management Group, *Deployment and Configuration Adopted Submission*, OMG Document ptc/03-07-08 edition, July 2003.
19. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, Getting started with AspectJ, *Commun. ACM* **44**(10) (2001) 59–65.
20. P. Tarr and H. Ossher and W. Harrison and S. M. Sutton, *N* degrees of separation: Multi-dimensional separation of concerns, in *Proc. Int. Conf. on Software Engineering*, May 1999, pp. 107–119.
21. D. Batory, J. Sarvela, and A. Rauschmayer, Scaling step-wise refinement, *IEEE Trans. on Software Engineering* **30**(6) (2004) 355–371.
22. P. Clemente, J. Hernandez, J. Herrero, J. Murillo, and F. Sanchez, Aspect-orientaton in the software lifecycle: Fact and fiction, in *Aspect-Oriented Software Development*, eds. R. Filman, T. Elrad, M. Aksit, and S. Clarke, Addison-Wesley, Reading, MA, 2004, pp. 407–424.
23. A. Rashid, A. Moreira, and J. Araujo, Modularisation and composition of aspectual requirements, in *Proc. Second Int. Conf. on Aspect-Oriented Software Development* (*AOSD*), Boston, MA, March 2003, pp. 112–120.
24. S. Clarke and E. Baniassad, *Aspect-Oriented Analysis and Design*, Addison-Wesley, 2005.
25. S. Clarke and R. J. Walker, Composition patterns: An approach to designing reusable aspects, in *Proc. Int. Conf. on Software Engineering* (*ICSE*), Toronto, Canada, May 2001, pp. 5–14.
26. R. B. France, I. Ray, G. Georg, and S. Ghosh, Aspect-oriented approach to early design modelling, *IEE Proceedings — Software* **151**(4) (2004) 173–186.
27. T. Elrad, O. Aldawud, and A. Bader, Expressing aspects using uml behavioral and structural diagrams, in *Aspect-Oriented Software Development*, eds. R. Filman, T. Elrad, M. Aksit, and S. Clarke, Addison-Wesley, Reading, MA, 2004, pp. 459–478.
28. Aspect-Oriented Software Development Conference, Bonn, Germany, *Eighth International Workshop on Aspect-Oriented Modeling*, March 2006.
29. M. Tkatchenko and G. Kiczales, Uniform support for modeling crosscutting structure, in Lecture Notes in Computer Science, *Proc. 8th Int. Conf. Model Driven Engineering Languages and Systems* (*MODELS 2005*), Montego Bay, Jamaica, October 2005, Springer Verlag, pp. 508–521.