International Journal of Software Engineering and Knowledge Engineering
World Scientific Publishing Company

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,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,zhangj,gray*}*@cis.uab.edu*

Domain-specific models increase the level of abstraction used to develop large-scale component-based systems. Model-driven development approaches (e.g., Model-Integrated Computing and Model-Driven Architecture) emphasize the use of models at all stages of system development. Decomposing problems using pure model-driven approaches, however, sometimes results in a separation of the artifacts in a way that impedes comprehension. For example, a single concern (such as deployment of distributed systems) may consist of different orthogonal activities (such as component specification, interaction, packaging and planning). From the perspective of keeping track of all entities associated with a component to ensure that the constraints for the system as a whole are not violated, a model-driven approach to describing the same system results in extra effort.

This paper provides three contributions to the study of applying aspect-oriented techniques to address the challenges of model-driven component-based distributed systems development outlined above. First, it evaluates the crosscutting concerns that arise in model-driven distributed systems development in the context of a domain-specific modeling language called the Platform-Independent Component Modeling Language (PICML). Second, it describes how aspect-oriented model weaving helps modularize the crosscutting concerns of component-based distributed systems using model transformations. Third, it describes how we have applied model weaving using a tool called the Constraint-Specification Aspect Weaver (C-SAW). 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 distributed systems development.

Keywords: Aspect-oriented model-weaving; Component-based development; Domain-

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

specific models.

1. Introduction

Model-driven development (MDD) is emerging as a new paradigm to develop complex distributed real-time and embedded (DRE) 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 (1) focusing on building individual components to (2) 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 (1) emphasize a visual approach to system development, which is crucial to composition and integration activities, (2) focus on describing system constraints using constraint languages [4], which can be enforced during design-time to prevent common errors that may otherwise occur late in the integration stage, (3) make the task of system analysis easier by providing better abstractions and notations closer to the domain of the system, and (4) shield system developers from changes in the underlying middleware platforms due to the increased level of abstraction.

MDD Challenges. Although MDD approaches are desirable in large-scale DRE system development, the promotion of modeling elements 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]. These concerns typically stem from the use of modeling abstractions to describe entities that were captured at the level of implementation in prior approaches. 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 the benefits offered by MDD approaches. What is desired, therefore, is an enhanced MDD approach that is (1) scalable with the number of modeling elements and the dependencies between them and (2) gives assurances that changes to properties of individual model elements can be performed non-intrusively.

Solution approach → Aspect-Oriented Model Weaving. Aspect-oriented model weaving [6] is a promising approach for addressing the problems associated with applying MDD-based approaches to large-scale distributed systems development. Aspect-oriented model weaving unites the ideas of aspect-oriented software development (AOSD) [7] with MDD to provide better modularization of model properties that crosscut multiple layers of a model [5].

Our approach to improving the scalability of MDD - and subsequently untangling the crosscutting concerns at the modeling level - relies on enhancing

the *Platform-Independent Component Modeling Language* (PICML) [8] by applying aspect-oriented model-weaving provided by the *Constraint-Specification Aspect Weaver* (C-SAW) [9]. PICML is an open-source domain-specific modeling language (DSML) (available for download at `www.dre.vanderbilt.edu/cosmic`) developed using the Generic Modeling Environment (GME) [10]. PICML enables developers of component-based DRE systems to define component interfaces, along with their properties and system software building rules, and 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 abstraction level using the *Embedded Constraint Language* (ECL). C-SAW assists modelers 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 from 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 DRE systems by using an unmanned air vehicle (UAV) application as a running example; Section 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; Section 4 compares our work with other tools that apply aspect-oriented approaches to distributed component systems development; and Section 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 software systems development [11]. 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 approaches to address these challenges.

In order to better illustrate the various challenges with MDD, we first present a brief overview of MDD approaches to developing component-based systems. We then illustrate an emergency response system, which uses multiple unmanned air vehicles (UAVs) to perform aerial imaging, survivor tracking and damage assessment. The UAV will serve as a motivating example to describe how a MDD solution can be applied to all stages of development. We then highlight the scalability challenges and crosscutting concerns a systems modeler faces when building a system like the

emergency response system.

2.1. *Overview of Model-driven Development of Component-based Systems*

In MDD, models are used to describe all artifacts of the system, i.e., 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, we advocate building a *domain-specific modeling language* (DSML), which can be viewed as a five-tuple [12] 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→C)**, which assigns syntactic constructs, e.g., graphical and/or textual) to elements of the abstract syntax,
- **Semantic mapping ($\mathbf{M}_S$: A→S)**, which relates the syntactic concepts to those of the semantic domain.

To support effective design and development of component-based systems, we have developed a DSML called PICML [8], which defines the different types of modeling elements that are essential to developing, 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) [13, 14], which was developed in collaboration with colleagues at Washington University.

2.2. *A Representative DRE System*

We now present a DRE system – an emergency response system – as the guiding example to illustrate the MDD approach, and to illustrate the challenges that arise in modeling these systems. This system is designed for 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 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 Figure 1 and described in detail in [8].

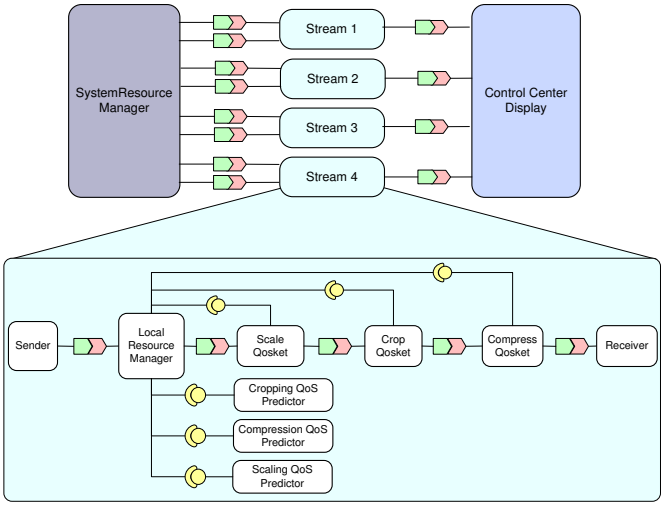

Fig. 1. Emergency Response System components

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` (i.e., 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` [13] components that 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. *Challenges in Applying MDD to an Emergency Response System*

This section describes how the PICML-based MDD approach is applied to the emergency response system while simultaneously highlighting the different scalability challenges and crosscutting concerns incurred by MDD. We illustrate these challenges as they manifest themselves in each of the modeling stages of systems development.

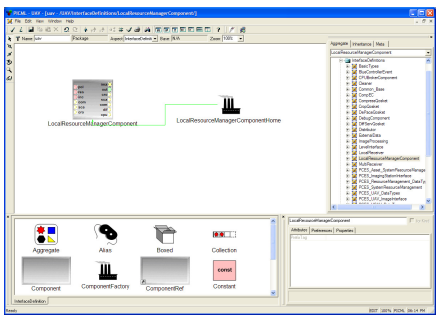

Fig. 2. Interface Definition

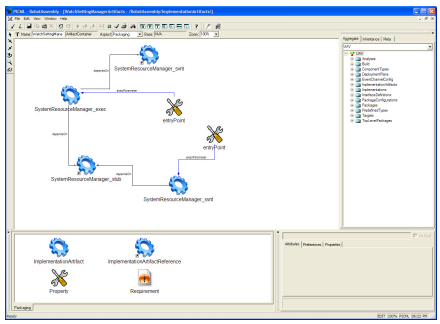

Fig. 3. Implementation Artifact Definition

2.3.1. *Crosscutting Concerns in Modeling Interface Definitions*

PICML allows modeling the individual component types in the 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 as shown in Figure 2. This step is very crucial, since the type (indicated by its name) of the individual monolithic components are 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.

These 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 modeler'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 modeler has to manually update/remove all the references scattered in the remaining model. This is an inherently time-consuming task, which is error-prone and doesn't scale if done manually.

2.3.2. *Modeling Implementation Artifact Definitions*

In this stage, a modeler defines the implementation artifacts shown in Figure 3 for each monolithic component, which involves defining the different implementation artifacts (e.g., shared libraries) that each component depends on, as well as describing the dependencies that each component may have on external system libraries.

For example, when building the UAV application using CIAO, a mono-

lithic component, say `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-related infrastructure.

Although the number, names and kinds of implementation artifacts might differ with the corresponding implementation, each component 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 which doesn't scale well. Moreover, the modeler is responsible for maintaining the dependencies between component instances and the dependent implementation artifacts. A mistake in the maintenance of the dependency will result in a run-time error due to unresolved dependencies of component instances on implementation artifacts.

Another example of tangled concerns arises from the need to follow specific naming conventions for the modeling elements, 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.

Yet another naming related problem is with the specification of the entry point for loading a shared library (as components are usually implemented). The modeler must ensure that the definitions of these entry points actually map to entry points defined in the shared libraries.

2.3.3. *Modeling Interaction Definitions*

In this stage of development, a modeler 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 which 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 Figure 4. 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.

Although there may be many component types defined in a model, it might be the case that only a subset of the component types need to be connected together to realize the application. For the components that are to be deployed, it is necessary to ensure that the associated implementation artifacts described in Section 2.3.2

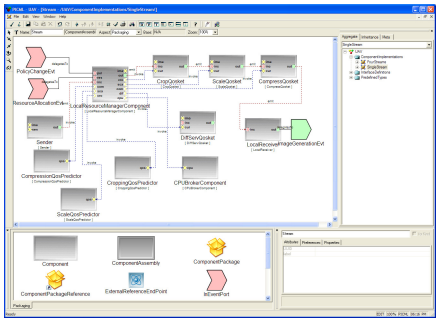

Fig. 4. Interaction Definition

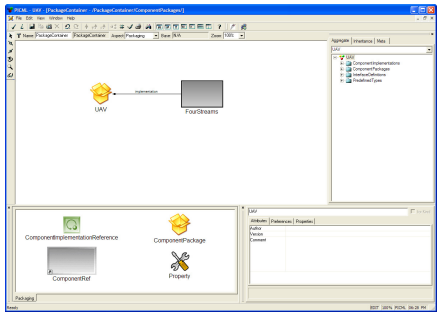

Fig. 5. Package Definition

are already defined, thus creating a dependency. It is also necessary that all the component instances that are defined in an assembly are also given appropriate mappings as explained in Section 2.3.5. Thus, any change to the assembly requires changes to a number of associated aspects, thereby highlighting the tangling of concerns. This is a tedious and error-prone task if done manually.

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. As shown in Figure 5, a package is associated with a top-level assembly, and is used to bootstrap the deployment of the application. 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 also serve the dual purpose of providing an entry point for bootstrapping the whole application. It is essential that an implementation of each component be available in such a form. It is necessary to ensure that this mapping between components and packages be maintained. It is also necessary that when changes to the assemblies are made, the appropriate packages are also updated. The need to ensure that changes to some elements are propagated to a dependent element is 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/or 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 need 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. This highlights both crosscutting concerns and scalability issues.

The use of MDD approaches to develop systems like the UAV application provides a significant improvement over programmatic approaches based on using only QoS-enabled component middleware. However, as outlined above, a number of tangled concerns and scalability issues manifest themselves in the modeling of DRE systems such as the UAV application. 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 challenges of modeling and developing large-scale distributed systems described in Section 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 Domain 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 [15], 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] as shown in Figure 6. For example, it is often the case that the meta-model 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" [16], 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

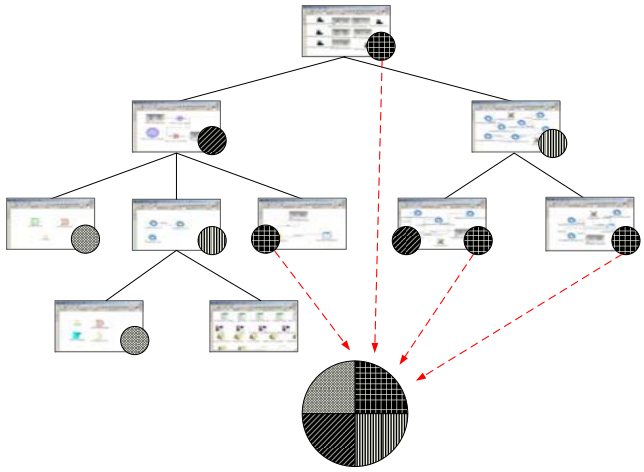

Fig. 6. Modularizing Crosscutting Concerns in Domain-specific Models

ability to specify properties across a model during the system modeling process. This action is performed by using a weaver that has been constructed with the concepts of modeling in mind.

3.2. *Aspect Modeling with C-SAW*

Our approach to AOM requires a domain-specific modeling weaver that processes the structured description of a visual model, which is different from traditional programming language weavers (e.g., the AspectJ weaver [15]) that support better modularization at a lower level of abstraction by processing source code.

We have designed C-SAW to provide support for modularizing crosscutting modeling concerns in the GME. This weaver operates on the abstract syntax tree of the model. 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. In general, the hooks registered with GME are specific to a particular DSML. 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 "plugin," a GME terminology for a DSML independent hook. Thus, the benefits of C-SAW are applicable across a whole spectrum of DSMLs.

To be effective, this weaver also requires the features of an enhanced constraint 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, forAll, exists, select). A unique feature of ECL that is not provided within OCL, however, is a set of reflective operators for navigating the hierarchical structure of a model. These operators can be applied to first class model objects (e.g., a container model or primitive model element) to obtain reflective information needed in AOM.

The AOM approach that we have adopted in C-SAW can be summarized by the diagram in Figure 7. As shown in this figure, 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.

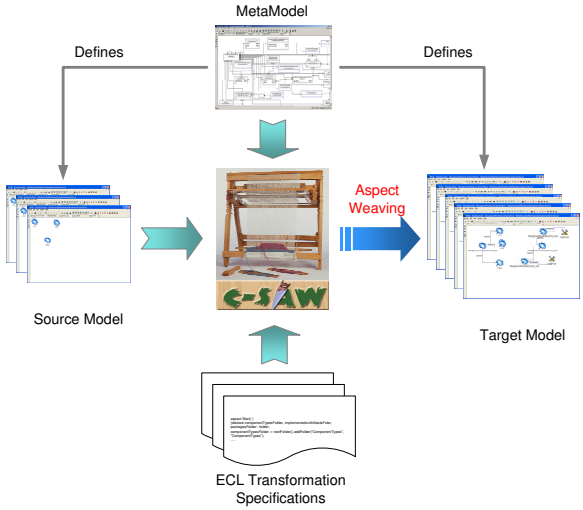

Fig. 7. C-SAW Aspect Model Weaver Framework

- **Modeling Aspect:** A modeling aspect is a modular construct that specifies a crosscutting concern across a model hierarchy. Each aspect describes the binding and parameterization of strategies to specific nodes in a model. A modeling aspect is responsible for identifying the specific locations of a crosscutting concern, and offers the capability to make quantifiable statements across the boundaries of a model.
- **Strategies:** A strategy is used to specify elements of computation, constraint

propagation, and the application of specific properties to the model nodes[a]. The name *strategy* is inspired by the strategy design pattern [17]. We use this term to define a collection of inter-changeable heuristics. Strategies are generic in the sense that their descriptions are not bound to particular model nodes. Each domain that supports a specific meta-level GME paradigm will have disparate strategies that can be applied to a model through C-SAW. Strategies provides a hook that the weaver can call to process node-specific constraint application and propagation. Strategies offer numerous ways for instrumenting nodes in the model with crosscutting concerns.

3.3. *Resolving UAV Crosscutting Modeling Challenges with C-SAW*

As described in Section 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 orthogonal artifacts that are necessary to deploy the UAV application.

```
aspect Deploy( )
{
  // Create a folder under the RootFolder called "ComponentTypes" of
  // kind ComponentTypes
  componentTypesFolder := rootFolder().addFolder("ComponentTypes", "ComponentTypes");

  // Create a folder under the RootFolder called
  // "ImplementationArtifacts" of kind ImplementationArtifacts
  implementationArtifactsFolder
    := rootFolder().addFolder("ImplementationArtifacts", "ImplementationArtifacts");

  // Create a folder under the RootFolder of kind ComponentPackages
  // called "Packages"
  packagesFolder := rootFolder().addFolder("ComponentPackages", "Packages");

  // Retrieve the componentAssembly folder, and generate deployment artifacts
  rootFolder().findFolder("ComponentImplementations").models()
    ->select(f | f.kindOf() == "ComponentImplementationContainer")
        ->models()->select(p | p.kindOf() == "ComponentAssembly")
          ->WeaveDeploymentArtifacts();
}
```

Aspect Listing 1: Deployment Specification Aspect

The task of modularizing the concerns of deployment begins with defining a *specification aspect* in C-SAW. Aspect Listing 1 shows a snippet of the definition of

[a] "model nodes" refer to modeling elements that are defined in the metamodel, and serve as visualization elements in the domain model

the `Deploy` aspect. This modeling aspect defines the tasks that a modeler typically performs manually. Specifically, 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. This aspect has been extended to cover all the different activities that were discussed in Section 2.3. Due to space constraints, we have not shown all of the different entities that are created by this aspect. The initial creation of these folders has a similarity to intertype declarations in AspectJ.

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. For each component assembly, it then applies the `WeaveDeploymentArtifacts` strategy as shown in Strategy Listing 1.

```
strategy WeaveDeploymentArtifacts()
{
  // Get the list of component instances within the assembly
  models()->select(c | c.kindOf() == "Component")->ImplementationArtifacts();
  models()->select(c | c.kindOf() == "Component")->PackageDefinition();
}
```

Strategy Listing 1: Weave Deployment Artifacts Strategy

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

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. It can be seen that this strategy modularizes:

- Creation of all implementation artifacts mandated by the underlying run-time,
- Creation of implementation artifacts which adhere to a specific naming convention,
- Keeping track of dependencies between a single monolithic component and it's associated implementation artifacts,
- Setting attribute values like location and entry points into shared libraries.

By modularizing the different activities associated with defining implementation artifacts and allowing for customizability based on idiosyncrasies of specific runtime environments, C-SAW helps resolve the challenge described in Section 2.3.2 by modularizing artifact definitions for all available components.

The `PackageDefinition` strategy shown in Strategy 3 is responsible for creation of a package and association of a component assembly with the component

```
strategy ImplementationArtifacts()
{
  component := self;
  componentName := component.getName();

  // Create an element of kind ArtifactContainer with the same name as the
  // component instance
  artContainer := rootFolder().findFolder("ImplementationArtifacts")
    .addModel("ArtifactContainer", componentName);

  // Create Foo_exec, Foo_stub and Foo_svnt
  ia_exec := artContainer.addAtom("ImplementationArtifact", componentName + "_exec");
  ia_stub := artContainer.addAtom("ImplementationArtifact", componentName + "_stub");
  ia_svnt := artContainer.addAtom("ImplementationArtifact", componentName + "_svnt");

  // Set the attribute "location" of Foo_exec, Foo_stub and Foo_svnt
  ia_exec.setAttribute("location", componentName + "_exec");
  ia_stub.setAttribute("location", componentName + "_stub");
  ia_svnt.setAttribute("location", componentName + "_svnt");

  // Create an element which is a reference to Foo_stub
  ia_stubRef
    := artContainer.addReference("ImplementationArtifactReference", ia_stub);

  // Create a connection of kind ArtifactDependsOn between Foo_svnt
  // and the Reference to Foo_stub
  artifactContainer.addConnection("ArtifactDependsOn", ia_svnt, ia_stubRef);

  // Create a connection of kind ArtifactDependsOn between Foo_exe
  // and the Reference to Foo_stub
  artifactContainer.addConnection("ArtifactDependsOn", ia_exec, ia_stubRef);
}
```

Strategy Listing 2: Implementation Artifact Strategy

```
strategy PackageDefinition()
{
  // Create an element of kind ComponentPackage with the same name as the
  // component instance (say Foo)
  compPackage := pkgContainer.addAtom("ComponentPackage", componentName);

  // Create an element of kind ComponentImplementationReference, which
  // references the MonolithicImplementation of the same name
  componentImplRef
    := pkgContainer.addReference("ComponentImplementationReference", monolithicImpl);

  // Create an element of kind ComponentRef, which references the type of
  // the component of Foo
  componentRef2 := pkgContainer.addReference("ComponentRef", component);

  // Create a connection between Foo -> Reference to implementation
  pkgContainer.addConnection("Implementation", compPackage, componentImplRef);

  // Create a connection between Foo -> Reference to component type
  pkgContainer.addConnection("PackageInterface", compPackage, componentRef2);
}
```

Strategy Listing 3: Package Definition Strategy

package. Similar strategies were defined to solve the challenges outlined in Section 2.3.1, Section 2.3.3 and Section 2.3.5. By combining the 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

A growing area of research is concentrated on bringing aspect-oriented techniques into the purview of analysis and design [18, 19]. A focal point of these efforts is the development of notational conventions that assist in the documentation of concerns that crosscut a design. These notational conventions advance the efficiency of expression of these concerns in the design. Moreover, they also have the important trait of improving the traceability from design to implementation.

Although these current efforts do well to improve the cognizance of AOSD at the design level, they generally tend to treat the concept of aspect-oriented design as an *adjective*. This is to say that the focus has been on the notational, semantical and decorative attributes concerned with aspects and their representation within UML. A contribution of this paper is to consider aspect-oriented 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 [20], which is focused on UML models.

There is an increasing interest among researchers toward applying advanced separation of concerns techniques to non-code artifacts [21]. In particular, AOSD techniques have been investigated at all levels of the development lifecycle [22], including requirements engineering and early design [23,24]. Several researchers have investigated the application of AOSD concepts within the context of the UML [25, 26]. These efforts have yielded guidelines for describing crosscutting concerns at higher levels of abstraction. In this regard, they have common goals with the work described in our paper. These efforts differ from our work, however, because we have been concentrating on the idea of building actual weavers for domain models.

VEST [27] is a toolkit that is built as a GME meta-model. It supports modeling and analysis of real-time systems and introduces the notion of prescriptive aspects to specify programming language independent advice to a design. A distinction between VEST and our C-SAW-based approach is in the generalizabilty of the weaving process. C-SAW is constructed to work with any GME meta-model (including VEST itself), while the strength of VEST lies in real-time system specification.

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, we describe 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. In particular, 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. Because of the crosscutting nature of model properties and constraints, the manual adaptation of a model becomes too error prone and hampers productivity because of all the mouse clicking and typing involved in each change.

We also show how we have successfully applied the C-SAW concepts to address the deployment aspect 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. *CORBA Components*, OMG Document formal/2002-06-65 ed., Object Management Group, 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. *Unified Modeling Language: OCL version 2.0 Final Adopted Specification*, OMG Document ptc/03-10-14 ed., Object Management Group, Oct. 2003.
5. J. Gray, T. Bapty, and S. Neema, "Handling Crosscutting Constraints in Domain-Specific Modeling," *Communications of the ACM*, pp. 87–93, Oct. 2001.
6. J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale, and B. Natarajan, "An approach for supporting aspect-oriented domain modeling," in *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE'03)*, Erfurt, Germany, Sept. 2003, pp. 151–168.
7. R. Filman, T. Elrad, M. Aksit, and S. Clarke, *Aspect-Oriented Software Development*. Reading, Massachusetts: Addison-Wesley, 2004.
8. 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 *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Francisco, CA, Mar. 2005, pp. 190–199.
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, Birmingham.
10. A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, Nov. 2001.
11. A. Gokhale, D. C. Schmidt, B. Natarajan, J. Gray, and N. Wang, "Model Driven Middleware," in *Middleware for Communications*, Q. Mahmoud, Ed. New York: Wiley and Sons, 2004, pp. 163–187.
12. G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development

of embedded software," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 145–164, Jan. 2003.

13. 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*, Q. Mahmoud, Ed. New York: Wiley and Sons, 2003, pp. 131–162.
14. N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian, "Configuring Real-time Aspects in Component Middleware," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, Agia Napa, Cyprus, Oct. 2004, pp. 1520–1537.
15. 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.
16. P. Tarr and H. Ossher and W. Harrison and S.M. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," in *Proceedings of the International Conference on Software Engineering*, May 1999, pp. 107–119.
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
18. "Sixth International Workshop on Aspect-Oriented Modeling," http://dawis.informatik.uni-essen.de/events/AOM_AOSD2005, Chicago, IL, March 2005.
19. S. Clarke and E. Baniassad, *Aspect-Oriented Analysis and Design*. Addison Wesley, 2005.
20. M. Tkatchenko and G. Kiczales, "Uniform support for modeling crosscutting structure," in *Proceedings of Sixth International Workshop on Aspect-Oriented Modeling*, March 2005.
21. D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Transactions on Software Engineering*, pp. 355–371, June 2004.
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*, R. Filman, T. Elrad, M. Aksit, and S. Clarke, Eds. Reading, Massachusetts: Addison-Wesley, 2004.
23. A. Rashid, A. Moreira, and J. Araujo, "Modularisation and composition of aspectual requirements," in *Proceedings of the Second International Conference on Aspect-Oriented Software Development (AOSD)*, Boston, MA, March 2003, pp. 112–120.
24. R. B. France, I. Ray, G. Georg, and S. Ghosh, "Aspect-oriented approach to early design modelling." *IEE Proceedings - Software*, vol. 151, no. 4, pp. 173–186, 2004.
25. S. Clarke and R. J. Walker, "Composition patterns: An approach to designing reusable aspects," in *Proceedings of the International Conference on Software Engineering (ICSE)*, Toronto, Canada, May 2001, pp. 5–14.
26. T. Elrad, O. Aldawud, and A. Bader, "Expressing aspects using uml behavioral and structural diagrams," in *Aspect-Oriented Software Development*, R. Filman, T. Elrad, M. Aksit, and S. Clarke, Eds. Reading, Massachusetts: Addison-Wesley, 2004.
27. 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.