Concern Separation for Adaptive QoS Modeling in Distributed Real-Time Embedded Systems

Jeff Gray[1], Sandeep Neema[2], Jing Zhang[3], Yuehua Lin[4],
Ted Bapty[2], Aniruddha Gokhale[2], Douglas C. Schmidt[2]

[1] *Dept. of Computer and Information Sciences, University of Alabama at Birmingham*
Birmingham AL 35294-1170
gray @ cis.uab.edu

[2] *Institute for Software Integrated Systems, Vanderbilt University*
Nashville TN 37211
{sandeep, bapty, gokhale, schmidt} @ isis.vuse.vanderbilt.edu

[3] *Motorola Research*
Schaumburg, IL 60196
j.zhang @ motorola.com

[4]*Honda Manufacturing of Alabama*
Lincoln, AL 35096
jane_lin @ hma.honda.com

Abstract

The development of distributed real-time and embedded (DRE) systems is often challenging due to conflicting quality-of-service (QoS) constraints that must be explored as trade-offs among a series of alternative design decisions. The ability to model a set of possible design alternatives—and to analyze and simulate the execution of the representative model—helps derive the correct set of QoS parameters needed to satisfy DRE system requirements. QoS adaptation is accomplished via rules that specify how to modify application or middleware behavior in response to changes in resource availability. This chapter presents a model-driven approach for generating QoS adaptation rules in DRE systems. This approach creates high-level graphical models representing QoS adaptation policies. The models are constructed using a domain-specific modeling language—the Adaptive Quality Modeling Language (AQML)—which assists in separating common concerns of a DRE system via different modeling views. The chapter motivates the need for model transformations to address crosscutting and scalability concerns within models. In addition, a case study is presented based on bandwidth adaptation in video streaming of unmanned aerial vehicles.

INTRODUCTION

The ability to adapt is an essential trait of distributed real-time and embedded (DRE) systems, where quality-of-service (QoS) requirements demand a system adjust to external evironment changes in a timely manner. A DRE system typically consists of a stack of software layers that are coupled to a physical system (e.g., a control system within an automotive factory or avionics subsystem). Several capabilities are needed to provide adaptability within DRE systems, including (1) the ability to express QoS requirements in some manipulatable form, (2) a mechanism to monitor important conditions associated with the environment and physical system, and (3) a causal relation between the mon-

itoring of the environment and the specification of the QoS requirements (Karr et al., 2001). Addressing QoS concerns via software adaptation has a concomitant effect on the physical system, such that a feedback loop is established between the physical parts of the system and the corresponding software.

Recent advances in middleware technologies have enabled a new generation of object-oriented DRE systems based on platforms such as Real-time CORBA and the Real-time Specification for Java (Dibble, 2008). Although middleware solutions promote software reuse resulting in higher development productivity (Schantz & Schmidt, 2001), provisioning and satisfying QoS requirements, such as predictable end-to-end latencies, remains a fundamental challenge for DRE systems, due to their distributed nature, unpredictable resource loads, and resource sharing. Moreover, specifying and satisfying QoS requirements in DRE systems often uses *ad hoc* problem-specific code optimizations, which impede the principles of reuse and portability.

To satisfy QoS requirements in the presence of variable resource availability, programmable QoS adaptation layers atop the middleware infrastructure have been proposed (Schantz et al., 2002). These QoS adaptation layers implement QoS adaptation policies, which include rules for modifying application or middleware behavior in response to a change in resource availability. The key idea behind such adaptation layers is to *separate concerns* with respect to QoS requirements. This separation provides improved modularization for separating QoS requirements from the functional parts of the software.

For example, the Quality Objects (QuO) project, developed by BBN, is an adaptation layer (Sharma et al., 2004) that extends the Object Management Group's (OMG) Interface Definition Language (IDL). This extension, known as the Contract Definition Language (CDL), is a textual language that supports the specification of QoS requirements and adaptation policies in the style of a state-machine. QuO contracts written in CDL are compiled to create stubs that are integrated into the QuO kernel and are used to monitor and adapt the QoS parameters when the system is operational.

Common QoS Implementation Problems

Although QuO's use of CDL works well from a software engineering perspective, there are drawbacks to using these approaches as the sole source for systems adaptation, including:

1. The control-centric nature of the programmatic QoS adaptation extends beyond software concepts, e.g., issues such as stability and convergence become paramount. In a DRE system, QoS is specified in software parameters, which have a significant impact on the dynamics of the overall physical system. Due to complex and non-linear dynamics, it is hard to tune the QoS parameters in an *ad hoc* manner without compromising the stability of the underlying physical system. The QoS adaptation software is, in effect, equivalent to a controller for a discrete, non-linear system. Sophisticated tools are therefore needed to design, simulate, and analyze the QoS adaptation software from a control system perspective.
2. Programmatic QoS adaptation approaches offer a lower level of abstraction, i.e., textual code-based. For example, implementing a small change to CDL-based adaptation policies requires manual changes that are scattered across large portions of the DRE system, which complicates ensuring that all changes are applied consistently. Moreover, even small changes can have far-reaching effects on the dynamic behavior due to the nature of emergent crosscutting properties, such as modifying the policy for adjusting communication bandwidth across a distributed surveillance system, as shown in the case study of this chapter.
3. QoS provisioning also depends on the performance and characteristics of specific algorithms that are fixed and cannot be modified, such as a particular scheduling algorithm or a specific communication protocol. These implementations offer fixed QoS and offer little flexibility in terms of tuning the QoS. Consequently, any QoS adaptation along this dimension involves structural adaptation in terms of switching implementations at run-time, which is highly complex, and in some cases infeasible without shutting down and restarting applications and nodes. Moreover, issues of state management and propagation, and transient mitigation, gain prominence amid such structural adaptations. Programmatic QoS adaptation approaches often offer little or no support for specifying such complex adaptations.

Sidebar 1. The Generic Modeling Environment

The Generic Modeling Environment (GME) (Lédeczi et al., 2001) is a domain-specific modeling tool that provides metamodeling capabilities that can be configured and adapted from meta-level specifications (representing the modeling paradigm) that describe the domain. GME provides a unified software architecture and framework for creating a customized domain-specific modeling environment (Balasubramanian et al., 2006; Lédeczi et al., 2001). The core components of the GME infrastructure are: a customizable *Generic Model Editor* for creation of multiple-view, domain-specific models; *Model Databases* for storage of the created models; and, a *Model Interpretation* technology that assists in the creation of domain-specific, application-specific model interpreters for transformation of models into executable/analyzable artifacts.

GME includes tools and functionality to support the creation and storage of system models, in addition to generation of executable/analyzable artifacts from these models. In many customizable modeling tools, including GME, the modeling concepts to be instantiated are specified in a *metamodeling* language (Karsai et al., 2004). A *metamodel* of the modeling paradigm is constructed that specifies the syntax, static semantics, and the presentation semantics of the domain-specific modeling paradigm. The metamodel uses a Unified Modeling Language (UML) class diagram to capture information about the objects that are needed to represent the system information and the relationships between different objects. The metamodeling language also supports the specification of visual presentation of the objects in the graphical model editor (Lédeczi et al., 2001).

A Solution: Aspects for Supporting Model-Driven Engineering

Addressing the challenges previously outlined requires raising the level of abstraction for reasoning about the systemic properties of DRE systems, including how the crosscutting QoS adaptations are effected and how they impact different parts of the DRE system. Model-Driven Engineering (MDE) (Bézivin et al., 2006; Schmidt, 2006) offers a promising approach to address these problems. From a modeling perspective, expressive power in software specification is gained from using notations and abstractions that are aligned to a specific problem domain (Mernik et al., 2005). This domain-specific approach can be further enhanced when graphical representations are provided to model the domain abstractions.

In domain-specific modeling, a design engineer describes a system by constructing a visual model using the terminology and concepts from a specific domain (Gray et al., 2007). Analysis can then be performed on the model and/or the model can be synthesized into an implementation. A key application area for MDE is in DRE systems that are tightly integrated between the computational structure of a system and its physical configuration (Sztipanovits and Karsai, 1997), such as the so-called cyber-physical systems typified by embedded computing. In such DRE systems, MDE has been shown to be a powerful paradigm for providing adaptability in evolving environments.

In this chapter, the challenges of QoS implementation are addressed by applying Aspect-Oriented Software Development (AOSD) (Kiczales et al., 1997; Tarr et al., 1999) techniques to MDE tools. Aspects have been effective at the programming language level to assist in modularizing concerns that were tangled within a single module, as well as other concerns scattered throughout multiple modules. Aspects provide a new construct to modularize such concerns in a manner not possible with the primary constructs in the paradigm of the development language, e.g., object-oriented languages such as Java cannot modularize many concerns that exhibit a crosscutting representation. Examples of crosscutting concerns at the implementation (i.e., source code) level include the canonical logging example, systems issues such as caching and prefetching, and web concerns like session expiration. The case study of this chapter demonstrates the application of aspects to MDE.

Figure 1 shows how MDE and aspects can be used to develop a DRE system. It involves the following three activities:

- **Design**. A domain-specific modeling tool is used to specify the structural composition and behavioral semantics of the DRE system. For example, as shown in Figure 1, the approach described in this chapter uses the *Generic Modeling Environment* (GME) to assist designers in modeling the DRE system and the QoS adaptation policy using standardized notations, such as Statecharts (Harel, 1987) and Dataflow (this is the Design phase of Figure 1). The modeling language is partitioned among various perspectives that allow model engineers to focus on specific related views of the design. Aspect-oriented weavers at the modeling level assist model engineers in rapidly changing crosscutting properties of a model that are traditionally hard to change due to the scattering of their specification. For example, a policy for

adapting bandwidth usage could span multiple models in a language whose semantics is based on a finite state machine (FSM), as demonstrated in the case study section of this chapter.

- **Synthesis**. The next stage is model transformation. MDE tools provide model interpreters associated with the domain-specific languages that can be used to traverse model instances to transform them into different kinds of artifacts, such as source code, input to analysis tools or metadata for configuration. In our example, as represented by the "Synthesis" part of Figure 1, a generator tool synthesizes artifacts for Matlab Simulink/Stateflow® (a popular commercial simulation tool), providing the ability to simulate and analyze the QoS adaptation policy. This tool enhances assurance that the system will perform as desired without having to deploy the actual system in the field. This generator tool has been implemented as a model interpreter for GME (Neema et al. 2002). The generator computes a mapping of the QoS adaptation policy represented in the modeling language into a semantically equivalent FSM representation in the Stateflow tool. In addition, the generator constructs a closed-loop control model in Simulink, in which the generated Stateflow model acts as the "controller," while the underlying computational system is abstracted as the "plant."

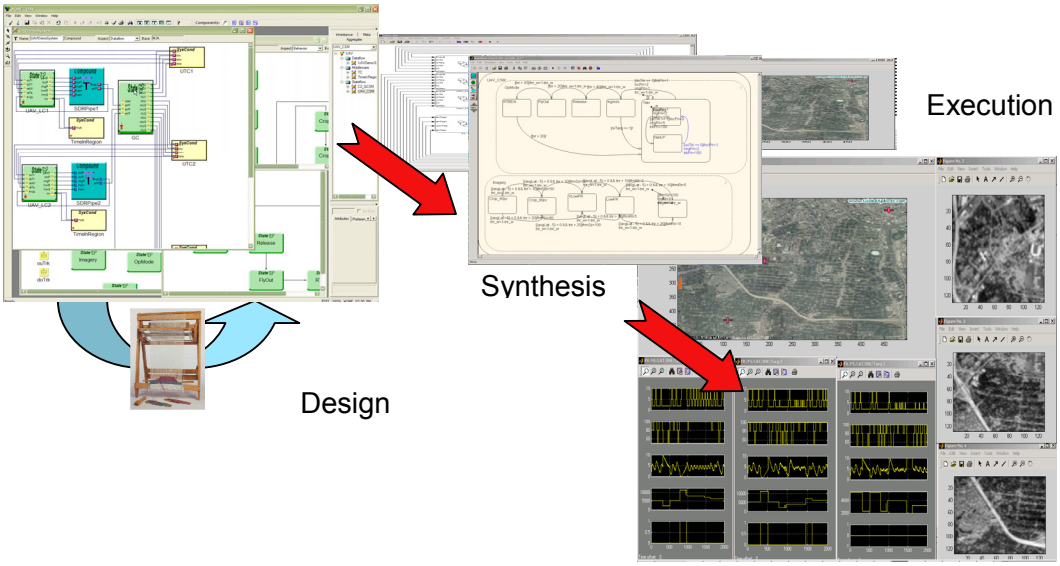


Figure 1. Model-Driven Design of Adaptive QoS for DRE Systems

- **Execution**. The final step is run-time QoS adaptation, which is performed via feedback. In our example, a second generator tool creates CDL specifications from the QoS adaptation models. The generated CDL is then compiled into executable artifacts (the "Execution" phase of Figure 1). The right-side of Figure 1 shows the screen-shots taken from the execution of a military target recognition system that was generated by a model representation using our approach. By modeling DRE systems at a higher level of abstraction, model engineers can evolve their designs without incurring the accidental complexities of specific implementation techniques. A previous study showed that small changes to an abstraction represented at the modeling level resulted in large changes across the corresponding CDL representation (Neema et al., 2002).

A growing body of literature is forming around the topic of aspect modeling and is perhaps best chronicled by the annual workshops on Aspect-Oriented Modeling (AOM) (www.aspect-modeling.org). A representative selection of publications in the AOM area can be found in (Reddy et al., 2006; Stein et al., 2006; Clarke & Baniassad, 2005), which focus on developing notational conventions that assist in documenting concerns that crosscut a design. These notational conventions advance the efficiency of expressing these concerns in the model. Moreover, they also have the important trait of improving traceability from design to implementation. The Motorola WEAVR (Cottenier et al., 2007) is similar to our approach, i.e., it is a real tool for aspect modeling, rather than simply a new notation. The Motoro-

la WEAVR only works with UML static diagrams, however, whereas our model weaver used in this chapter is applicable to any modeling language.

Although current efforts do well to improve the cognizance of AOSD at the design level, they generally tend to treat the concept of aspect-oriented modeling primarily as a specification convention. The focus has therefore been on the graphical representation, semantic underpinnings, and decorative attributes concerned with aspects and their representation within UML. A contribution of this chapter is to consider AOM more as an operational task by constructing executable model weavers, i.e., AOSD is used as a mechanism to improve the modeling task itself by providing the ability to quantify properties across a model throughout the system modeling process (Filman & Friedman, 2000). This action is performed by utilizing a model weaver that has been constructed with the concepts of domain-specific system modeling in mind. This chapter focuses attention on the Design phase of DRE systems, as shown in Figure 1. The approach described in this chapter has benefits that are similar to those of model-driven middleware (MDM) (Gokhale et al., 2008) and adaptive and reflective middleware (ARM) (Schantz and Schmidt, 2001). The primary contribution of the chapter is the utilization of an aspect-oriented weaver that performs model transformations across higher level abstractions to separate policy decisions that were previously scattered and tangled across the model.

VIEWPOINT MODELING FOR QoS ADAPTATION IN DRE SYSTEMS

The Adaptive Quality Modeling Language (AQML) is a GME-based modeling language that assists in modeling key aspects of a DRE system. Each area of concern in the model is partitioned into a specific view that slices through a particular perspective of the overall model. The separation of concerns provided by a GME view is similar in intent to previous research on viewpoints (Nuseibeh et al., 1994) in requirements engineering. AQML defines the following three views defined in the metamodel (Neema et al, 2002) as follows:

1. **QoS adaptation modeling**, which models the adaptation of QoS properties of the DRE system. Designers can specify the different state configurations of the QoS properties, the legal transitions between the different state configurations, the conditions that enable these transitions (and the actions that must be performed to enact the change in state configuration), the data variables that receive and update QoS information, and the events that trigger the transitions. These properties are modeled using an extended finite-state machine (FSM), which is useful for specifying actions in a control-centric environment. Other Models of Computation (MoC) may be useful in different contexts (e.g., queuing models are useful for performance estimation).
2. **Computation modeling**, which models the computational aspect of a DRE system. A Dataflow MoC is employed to specify the various computational components and their interaction. The Dataflow MoC is chosen because it is well-suited to a particular class of DRE systems, namely streaming distributed multimedia application, which is the focus of this chapter. It should be noted, however, that the approach of QoS adaptation presented in this chapter is general, and can be composed with other MoCs for computational modeling.
3. **Middleware modeling**, which models the middleware services, the system monitors, and the tunable "knobs" (i.e., the parameters being provided by the middleware) in such a way that the parameters are specified for possible analysis. This category of modeling ensures the many configuration parameters offered by middleware are used correctly, so the deployment of an application is optimal rather than suboptimal due to the selection of a collection of parameters that have opposing effects or should not be used together.

The next four sub-sections describe the metamodel of each of these modeling categories and their interaction in AQML. Each metamodel represents a view of the overall system and each metamodel is linked together through interactions.

QoS Adaptation Modeling

Stateflow models capture the adaptive QoS behavior of DRE systems. A discrete FSM representation, extended with hierarchy and concurrency, is provided for modeling the QoS adaptive behavior of the system. This representation has been selected due to its scalability, universal acceptability, and ease-of-use in modeling. Figure 2 shows the QoS adaptation view of AQML.

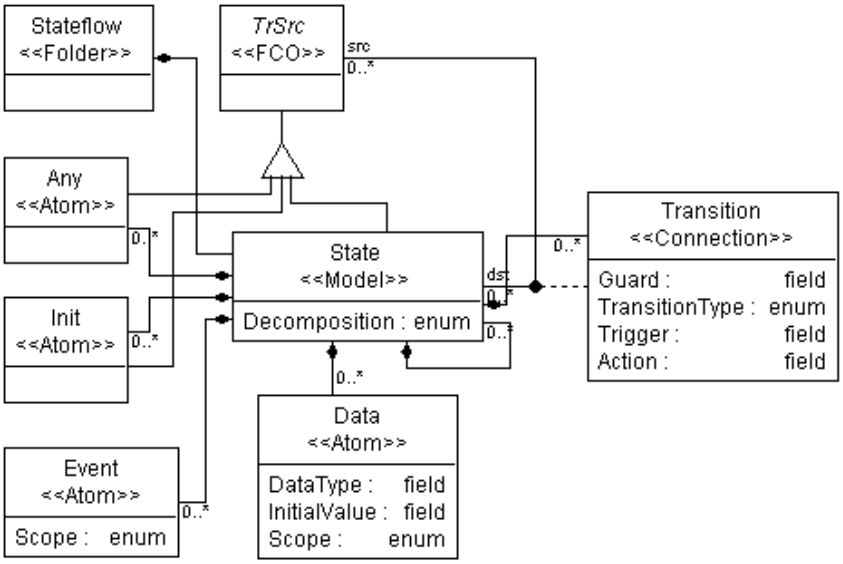


Figure 2. Metamodel of QoS Adaptation Modeling

The main concept in an FSM representation is a *State*, which defines a discretized configuration of QoS properties. Hierarchy is enabled in the representation by allowing States to contain other States. Attributes define the decomposition of the State. The State may be an *AND* state (when the state machine contained within the State is a concurrent state machine), or, the State can be an *OR* state (when the state machine contained within the State is a sequential state machine). If the State does not contain child States, then it is specified as a *LEAF* state. States are stereotyped as *models* in GME.

Transition objects are used to model a transition from one state to another. The attributes of the transition object define the trigger, the guard condition, and the actions. The trigger and guard are Boolean expressions. When these Boolean expressions are satisfied, the transition is enabled and a state change (accompanied with the execution of the actions) takes place. Transitions are stereotyped as a *Connection* in the GME tool. To denote a transition between two States, a connection has to be made from the source state to the destination state.

In addition to states and transitions, the FSM representation includes data and events. These can be directly sampled external signals, complex computational results, or outputs from the state machine. In the AQML, *Event* objects capture the Boolean event variables, and the *Data* objects capture arbitrary data variables. Both the Events and Data have a Scope attribute that indicates whether an event (or data) is either local to the state machine or is an input/output of the state machine.

Computation Modeling

This view is used to describe the computational architecture. A dataflow representation, with extensions for hierarchy, has been selected for modeling computations. This representation describes computations in terms of computational components and their data interactions. To manage system complexity, the concept of hierarchy is used to structure the computation definition. Figure 3 shows the computation view of the AQML.

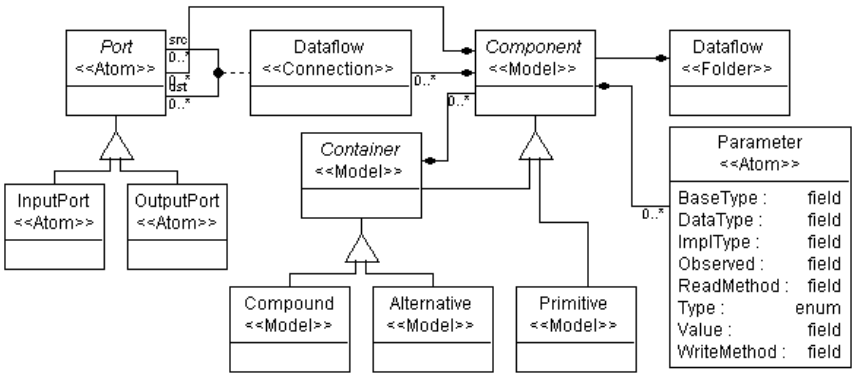


Figure 3. Metamodel of Computation Architecture Modeling

The computational structure is modeled with the following classes of objects: *Compounds*, *Alternatives*, and *Primitives*. These objects represent a computational component in a dataflow representation. *Ports* are used to define the interface of these components through which the components exchange information. Ports are specialized into *InputPorts* and *OutputPorts*.

A Primitive is a basic modeling element that represents an elementary component. A Primitive maps directly to a processing component that will be implemented as a software object or a function. A Compound is a composite object that may contain Primitives or other Compounds. These objects can be connected within the compound to define the dataflow structure. Compounds provide the hierarchy in the structural description that is necessary for managing the complexity of large designs. An Alternative captures "design choices" – functionally equivalent designs for a rigorously defined interface, providing the ability to model design spaces instead of a single design. The use of Alternatives allows capturing discrete combinatorial design spaces in a highly compact and scalable representation. The large design space thus modeled, however, must be explored to determine the subset of designs that satisfy requirements. An automated Design Space Exploration Tool (DESERT), assists the user in performing this exploration. A detailed description of DESERT is beyond the scope of this chapter; the interested reader is referred to (Neema et al., 2003).

An important concept relevant to QoS adaptive DRE systems is the notion of parameters, which are the tunable "knobs" used by the adaptation mechanism to tailor the behavior of the components so the desired QoS properties are maintained. Parameters can be contained in Compounds, Primitives, and Alternatives. The Type attribute defines whether a Parameter is read-only, write-only, or read-write. The DataType attribute defines the data type of the parameter.

Middleware Modeling

In this view, the concerns of the middleware are modeled, which include the services and the system conditions provided by the middleware. Example services include an Audio-Video Streaming service, Bandwidth reservation service, Timing service, and Event service. *System conditions* are components that provide quantitative diagnostic information about the middleware. Examples of these include observed throughput, bandwidth, latencies, and frame-rates. Figure 4 shows the middleware modeling view of the AQML.

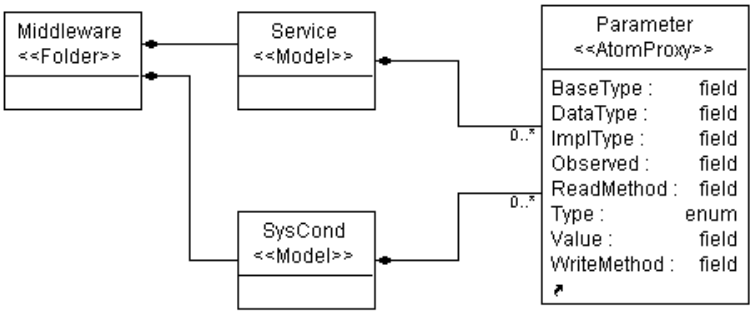


Figure 4. Metamodel of Middleware Modeling

The *Service* object represents the services provided by the middleware. Services can contain parameters that are the configuration points provided by the service. In addition to being tunable "knobs," parameters play a second role as instrumentation, or probes, by providing some quantitative information about the service. The *SysCond* object represents the system condition objects present in the middleware layer. SysConds can also contain parameters.

We do not facilitate a detailed modeling of the middleware components or the dataflow components because the focus of AQML is on the QoS adaptation. We model only those elements of the dataflow and middleware that facilitate the QoS adaptation (namely, the tunable and observable Parameters).

Interaction of QoS Adaptation with Middleware and Computation Modeling

This category specifies the interaction of the previous three modeling categories, as shown in Figure 5. As described earlier, the Data/Event objects within the Stateflow model form the interface of the state machine. Within the Computation and the Middleware views, Parameters form the control interfaces. The interaction of the QoS adaptation (captured in Stateflow models), and the middleware and application (modeled in the Middleware/Computation models), is through these interfaces. The interaction is modeled with the *Control* connection class, which connects the Data object of a State model to a Parameter object of a Middleware/Computation model.

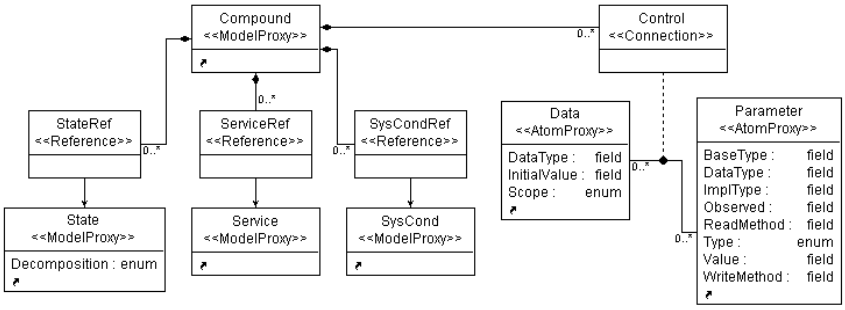


Figure 5. Interaction Metamodel of QoS Adaptation with Middleware/Computation Modeling

In GME, connections between objects that are not contained within the same model are specified as references, which are equivalent to a "reference" or a "pointer" in a programming language (Karsai et al, 2004). These are contained in the same context (model) such that a connection can be drawn between them. The *StateRef*, *ServiceRef*, and the *SysCondRef* objects are the reference objects that are contained in a Compound (Computation) Model. All the views mentioned in this section partition the concerns of a larger model space into cohesive units that make the modeling activity more manageable.

In addition to the physical interactions specified in the metamodel of Figure 5, there are deeper and indirect interactions across the different views that are not expressed in the models, yet must be accounted for by the designer of the QoS adaptations. The aggregate QoS space of the system is in essence a subset of the cross product of the (potentially continuous and infinite) state-space of the Computation, the Middleware, and the Resource. The states in the QoS Adaptation aspect represent an abstraction and partitioning of the QoS space into discrete states. Similarly, there are linkages between Computation, and Middleware, through direct utilization of APIs, scheduling of resources, and communication. In the approach presented, some of these interactions must be accounted for in the plant model (e.g., when simulating the QoS adaptation policies). In general, however, QoS adaptation designers must consider these latent interactions when designing the QoS adaptation policies.

The three metamodels defined in this section correspond to specific views of a DRE system. At the instance model level, this multidimensional view perspective provides model engineers with multiple perspectives for separating different concerns of the model such that details not pertinent to the modeling task at hand are abstracted into other views. Although this approach offers a valuable modeling construct, viewpoints alone are not sufficient for capturing certain types of crosscutting concerns, as shown later in this chapter. Before describing that issue, however, the next section introduces the problem domain used in the case study of the chapter, which is built as an instance of the AQML metamodel.

SPECIFYING QoS POLICIES FOR ADAPTATION OF VIDEO BANDWIDTH

The case study used in this chapter demonstrates an application of QoS modeling as applied to a conceptual scenario involving a number of Unmanned Aerial Vehicles (UAVs) conducting surveillance, e.g., in support of disaster relief efforts. Each UAV streams video back to a central distributor that forwards the video on to several different displays (Loyall et al., 2001). The feedback cycle for utilizing UAVs as surveillance devices includes (1) video from each UAV is sent to the distributor that is located in a command center located on a ground station, (2) the distributor broadcasts the video to numerous video display hosts at the command center, (3) the video is received by each host and displayed to various operators, and (4) each operator at a display observes the video and sends commands, when deemed necessary, to control the UAVs (Karr et al., 2001). The UAV prototype used for our case study was developed by researchers from BBN as an integration testbed and experimental platform for the DARPA Program Composition for Embedded Systems (PCES) project.

The concept of operation for the scenario can be summarized briefly as follows. Initially, several UAVs are conducting surveillance in a region of interest. UAV imagery must be transmitted in real-time to ground stations, ensuring an acceptable image quality. The latency of the transmission of an image and its reception at the ground station must be low enough to provide a continuous update of the region of interest as the UAVs loiter above. The available bandwidth must be shared uniformly over all of the collaborating UAVs and each UAV must adapt its transmission rate to the available bandwidth such that the timeliness requirement of the delivered imagery is met.

The scenario advances to the next stage when one or more of the UAVs observe a target in their field-of-view. In this next stage of the scenario, QoS requirements evolve as the UAVs observing the target acquire primary importance with respect to the collective goals of the system. It is required that the UAVs observing a potential target receive preferential treatment in bandwidth distribution such that the transmission of the critical information they observe is not delayed. All other UAVs that are not observing a target must reduce their bandwidth usage. In this case study, attention is restricted to just these two stages of the scenario in order to keep the description comprehensible (although there are additional stages that exercise a broader spectrum of QoS adaptation).

In the presence of changing conditions in the environment, the fidelity of the video stream must be maintained according to specified QoS parameters. The video must not be stale, or be affected by jitter to the point that operators cannot make informed decisions. Within the BBN implementation of the QuO project, a *contract* assists the system developer in specifying QoS requirements that are expected by a client and provided by a supplier. Each contract describes operating regions and actions that must be taken when QoS measurements change.

A textual domain-specific language (DSL) was developed by BBN to assist in the specification of contracts; the name of this DSL is the *Contract Description Language* (CDL) (Karr et al., 2001; Schantz et al., 2002). A code generator translates the CDL into code that is integrated within the run-time kernel of the application. The textual intention of a CDL specification is similar to the semantics of a hierarchical state machine. The overall approach adopted by BBN for implementation of QoS adaptation is aspect-oriented (Duzan et al., 2004).

Several things make the UAV case study a complex and challenging problem, including its real-time requirements, resource constraints, and distributed nature. In addition, there are other interesting observations to consider, including (1) the link between the UAVs to the host command center is a wireless link imposing some strict bandwidth restrictions, (2) there is a need for prioritization between different video streams coming from different UAVs owing to the region of interest, (3) latency is a higher concern than throughput because it is important to get the latest changes in the threat scenario at the earliest possible time, (4) there may be a wide-variety of computational resources (e.g., processors, networks, switches) involved in the entire application, and (5) the scenario is highly dynamic, because UAVs frequently enter and leave regions of interest.

To address these complex requirements, a QoS-enabled middleware solution has been designed for this application (Sharma et al., 2004). AQML was developed in response to the need for a modeling language to represent QoS specification for the UAV application (Neema et al., 2002). The next sub-section introduces some instances of the AQML metamodel applied to the UAV case study.

AQML Modeling of UAV Interactions

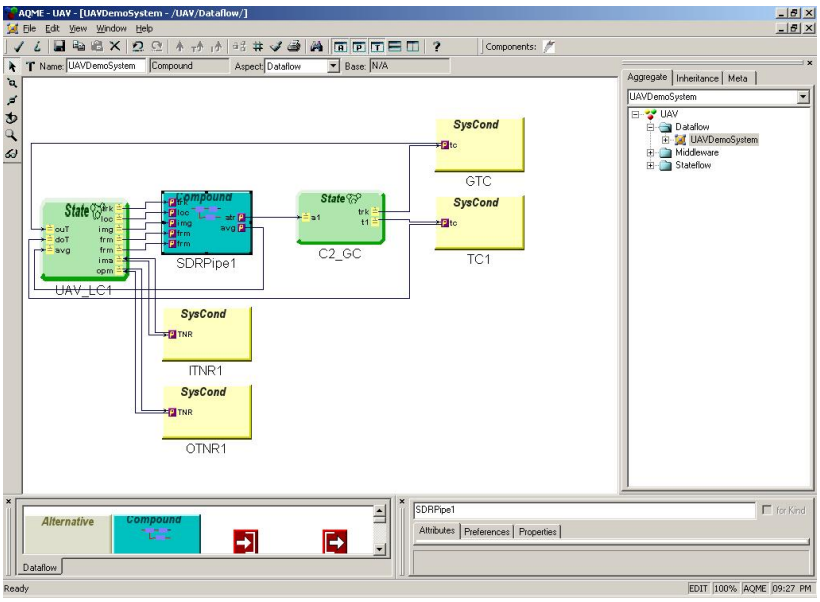


Figure 6. Top-Level Computational Components for One UAV

Figure 6 shows the top-level computational components for one UAV and its interactions. The view shown is a dataflow diagram with interactions between the UAV adaptation controller, the middleware system condition variables, and the computational components responsible for transmission of video content. This view represents the integration of the different concerns (i.e., Qos Adaptation, Application Computation, and Middleware), which are captured in separate diagrams and merged together in the integrated view. The box labeled SDRPipe1 represents the top-level of the hierarchical composition of the Sender-Distributor-Receiver components that are responsible for production, distribution, and display of the video stream. The UAV_LC1 component represents the top-level of the UAV QoS adaptation controller (detailed further in the following). The C2_GC component represents the top-level of the Ground Station's QoS adaptation controller, which is responsible for coordinating between UAVs for distribution of bandwidth according to the tactical importance of each UAV.

This particular view reveals only a single UAV. The OTNR1, ITNR1, GTC, and TC1 are system condition variables, which are middleware objects responsible for communicating QoS information across different objects in a distributed system. The OTNR/ITNR represents time in region, which keeps track of the time the UAV has been in a particular mode of operation. The TC1 and GTC keep track of the observation of a target by the UAVs. The TC1 is a variable set by UAV1 when it sees a target. The GTC is a logical-OR of all the TC variables, indicating if one or more of the UAVs have observed a target. All the lines shown in the view represent flow of information among these components. The UAV_LC1 controls the video stream parameters such as frame rate, frame size, and image quality, which is indicated by the connections between the UAV_LC1's and the SDRPipe1's appropriately named ports.

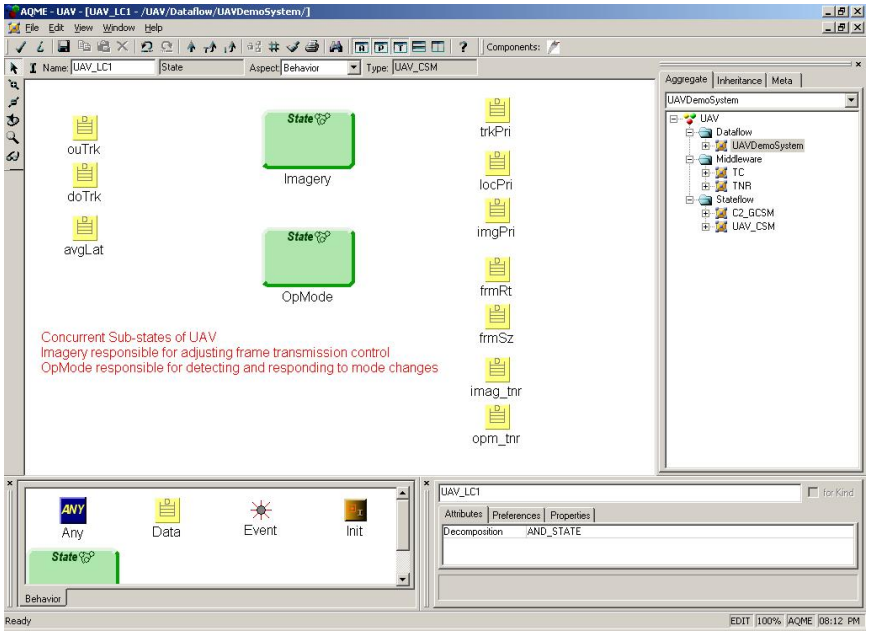


Figure 7. Concurrent States within one UAV

Figure 7 and Figure 8 show a model of the QoS-adaptive behavior of the UAVs as modeled in AQML. Figure 7 shows the top-level behavior consisting of two concurrent states: Imagery and OpMode. The OpMode state represents the operational modes of the UAV related to the observation of a target. The Imagery state manages the adaptation of imagery transmission based on the available bandwidth and the operational mode. Figure 8 shows the sub-states of the Imagery state.

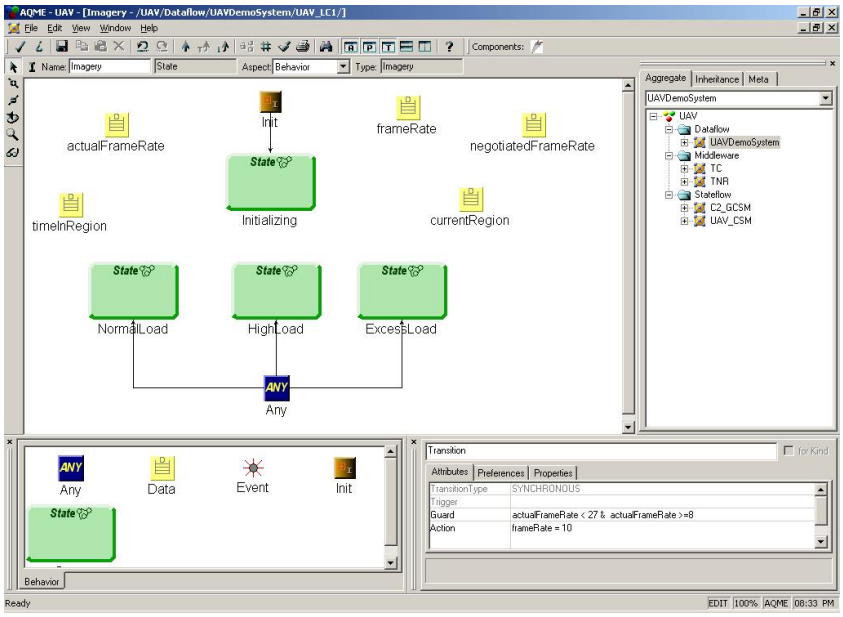


Figure 8. Model of QoS Adaptation within Imagery State

In this application, the goal of QoS adaptation is to minimize the latency on the video transmission. When the communication resources are nominally loaded, it may be possible to transmit the video stream at the full frame rate with a minimal basic network delay. When the communication and computational resources are loaded, however, the delays in transmission expand for the same frame rate resulting in increased latencies. The adaptation scheme attempts to compensate for the increased load by reducing the rate of transmission, thus improving the latency again.

There are several ways to reduce the transmission rate, including (1) reducing the frame rate by dropping frames, (2) reducing the image quality per frame, or (3) reducing the frame size. Depending on the actual scenario, one or more of these situations may apply. In the example of this chapter, dropping the frame rate is the only option considered, but other alternatives are possible, e.g., changing the video from color to black and white.

Figure 8 shows a QoS adaptation model of the UAV scenario in the AQML. The three states NormalLoad, ExcessLoad, and HighLoad capture three different QoS configurations of the system. A few data variables (actualFrameRate, frameRate, timeInRegion) can also be seen in this figure. These data variables provide the state machine with sensory information about the network. At the same time, other data variables may enact the adaptation actions that are being performed in the transitions.

The attribute window at the bottom-right corner of Figure 8 shows the trigger, guard, and action attributes of a transition. An example guard expression is visible in the attribute window of the figure (i.e., "actualFrameRate < 27 and actualFrameRate >= 8"). When this expression evaluates to true, the transition is enabled and the modeled system enters the HighLoad state. An example action expression can be seen in this figure (i.e., "frameRate = 10"). This sets the frameRate data variable to a value of 10.

The next section provides a UAV case study modeled in AQML that uses model transformations to address challenges of modeling that frequently occur in the modeling process. The case study highlights the benefits that model transformations provide to adapt a DRE system in the presence of crosscutting concerns and scalability requirements.

TRANSFORMATIONS FOR RAPID EVOLUTION OF MODELS

Although viewpoints provide a valuable mechanism for managing disparate concepts within a design, these views are not without interactions. Designers are typically responsible for maintaining consis-

tency among views. Moreover, maintaining this consistency is a non-trivial task as the system evolves. Aspect modeling offers a mechanism to automate these interactions.

The Constraint-Specification Aspect Weaver (C-SAW) is a model transformation engine that unites the ideas of aspect-oriented software development (AOSD) (Kiczales et al., 1997; Tarr et al., 1999) with MDE to provide better modularization of model properties that are crosscutting throughout multiple layers of a model (Gray et al., 2001; Gray et al., 2004). In the same manner that code can be scattered across the boundaries of source modules, the same emergent behavior occurs at the modeling level. For example, the addition of a black box data recorder into the model of a mission computing avionics system requires modeling changes across the whole collection of model entities (Gray et al., 2006). C-SAW is available as a GME plug-in and provides the ability to explore numerous modeling scenarios by considering crosscutting modeling concerns as aspects that can be inserted and removed rapidly from a model. The next two sections provide examples of transformations that address the separation of crosscutting modeling aspects and scalability issues within large models for DRE systems.

Within the C-SAW infrastructure, the language used to specify model transformation rules and strategies is the Embedded Constraint Language (ECL), which is an extension of the Object Constraint Language (OCL). ECL 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). It also provides special operators to support model aggregates (e.g., models, atoms, attributes), connections (e.g., source, destination) and transformations (e.g., addModel, setAttribute, removeModel) that provide access to modeling concepts that are within GME.

There are two kinds of ECL specifications: (1) a modeling specification that describes the binding and parameterization of strategies to specific entities in a model and (2) a strategy that specifies elements of computation and the application of specific properties to the model entities.[1] C-SAW interprets these specifications and transforms the input source model into the output target model. The C-SAW web site (www.cis.uab.edu/gray/Research/C-SAW) is a repository for downloading papers, software, and several video demonstrations that illustrate model transformation with C-SAW in GME. The following sections provide representative examples of ECL and the concept of aspect model weaving and model transformation applied to behavioral modeling of embedded systems.

Weaving Across Finite State Machines

When writing a specification for QoS adaptation, there typically arises one dimension of the adaptation that is treated as a dependent variable. There are numerous other independent variables that are adjusted to adapt the dependent variable according to some QoS requirement. For example, the end-to-end latency of video stream distribution may be a dependent variable that drives the adaptation of other independent variables (e.g., the size of a video frame, or even the video frame rate). In such cases, a *policy* is defined that represents the process for performing QoS adaptation.

The actions prescribed by a policy often crosscut the structure of a hierarchical state machine. Changing the policy requires modifying each location of the state machine that is affected by the policy. Elrad et al. have also reported on scenarios where state machine models are crosscutting (Elrad et al., 2002), but their approach is notational in nature and does not utilize a weaver at the modeling level. The Motorola WEAVR (Cottenier et al., 2007) represents one of the most mature aspect modeling implementations. The WEAVR is also focused on state machines, but is limited in application to UML. The weaving described in this chapter can be integrated with other modeling languages.

The C-SAW weaver has been applied to the AQML language to assist in weaving properties according to the semantics of an adaptation policy. Several strategies have been created to support the modeling of state machines that represent the behavior of a contract. The first strategy focuses on issues related to the creation of state machines and their internal transitions. The view of the model shown in Figure 9 pertains to the dataflow of the UAV case study.

[1] More details about ECL are available in (Lin, 2007).

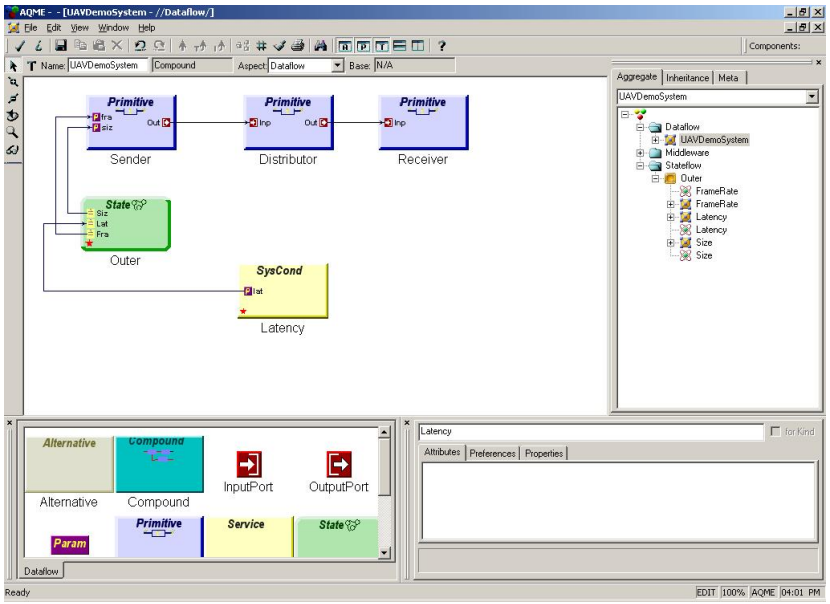


Figure 9. Dataflow for UAV Prototype

The model in Figure 9 is an instance of the AQML metamodel. In this case, the latency concern is the dependent variable that represents a system condition object whose value is monitored from the environment. The latency is an input into a hierarchical state machine called Outer. Within Outer, there are internal state machines that describe the adaptation of identified independent control variables (e.g., FrameRate and Size, as shown with the dependent Latency variable in Figure 10).

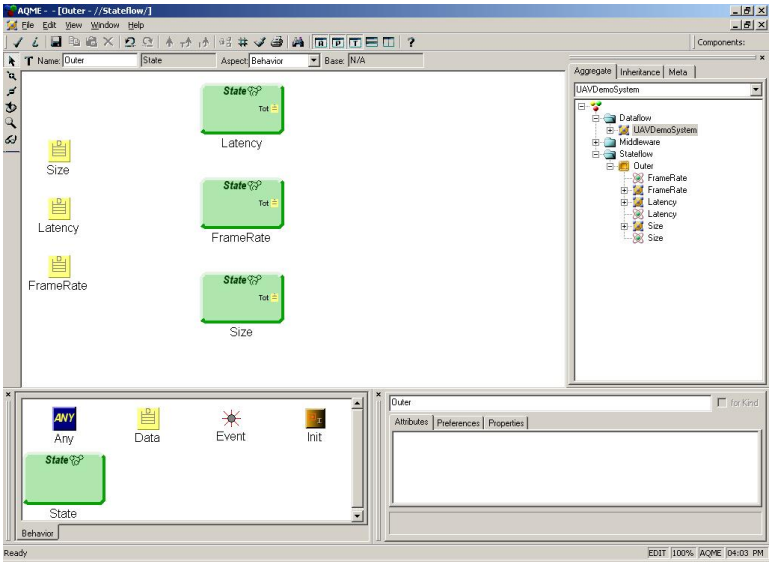
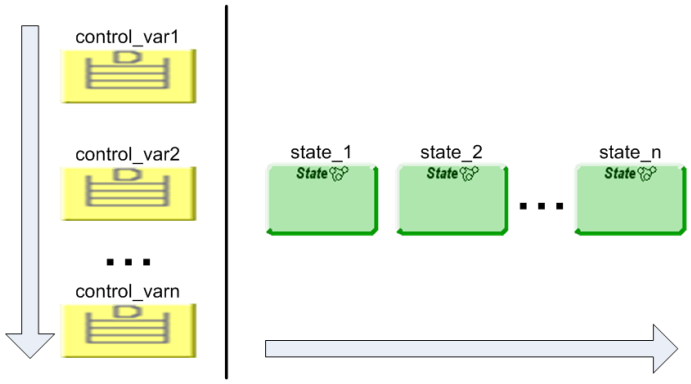


Figure 10. Top-Most View of Parallel State Machine

As shown in Figure 11, there are two ways that a state machine model can be extended to address QoS adaptation. Along one axis of extension, the addition of new dependent control variables often can offer more flexibility in adaptation toward the satisfaction of QoS parameters. It could be the case that other variables in addition to FrameRate and Size would help in reducing the latency, e.g., color, video format, and compression. Figure 11a captures the intent of this extension by introducing new control variables. It may also be the case that finer granularity of the intermediate transitions within a

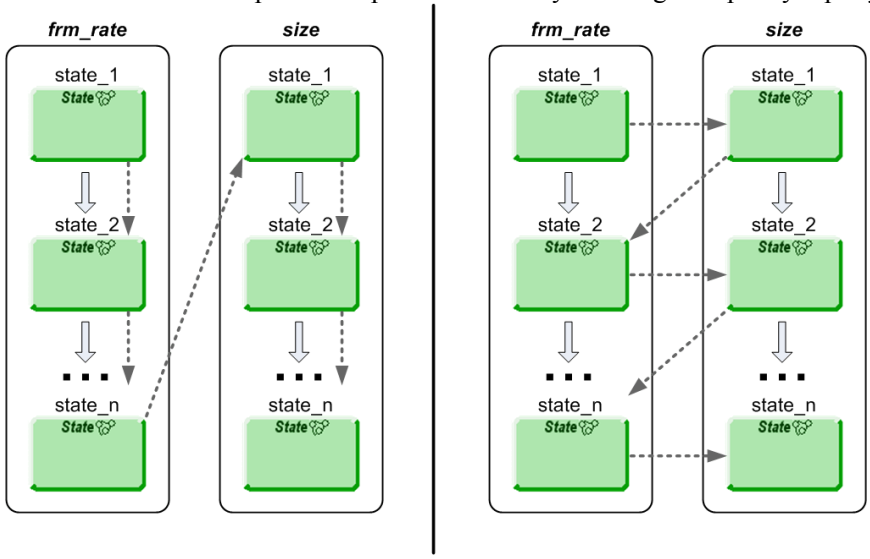
particular state would permit better adaptation to QoS requirements. Figure 11b captures the intent of this extension.



(a) Adding New Control Variables (b) Adding More Intermediate Transitions in States

Figure 11. Axes of Variation within a State Machine

In addition to the strategy for creating control variables and their intermediate states, an additional strategy was written to provide assistance in changing the adaptive policy that spans across each state machine. There could be numerous valid policies for adapting a system to meet QoS requirements. Two possibilities are shown in Figure 12. The realization that each of these policies is scattered across the boundaries of each participating state machine suggests that these protocols represent a type of crosscutting concern that should be separated to provide an ability to change the policy rapidly.



(a) Priority Exhaustive (b) Zig-zag

Figure 12. Policies for Adapting to Environment

The left side of Figure 12 specifies a protocol that exhausts the effect of one independent variable (frm_rate) before attempting to adjust another independent variable (size). The semantics of this protocol pertain to the exhaustive reduction of one variable before attempting to reduce another one. The size variable is therefore of a higher priority in this case because it is not reduced until there is no further reduction possible to the frame rate. The dotted-arrow in this figure indicates the order in which the transitions fire based upon the predicate guards.

The right side of Figure 12 represents a more equitable strategy for maintaining the latency QoS requirement. In this protocol, a zig-zag pattern suggests that the reduction of a variable is staggered with the reduction of a peer variable. Observe that Figure 12 involves only two control variables. The ability to change the protocol (by hand) becomes complicated when many variables are

involved, or when there are numerous intermediate states. This crosscutting nature suggests that a C-SAW strategy would be beneficial to assist in the exploration of alternative policies. Figure 13 contains a strategy that supports the protocol highlighted in Figure 12a.

```
1    defines AddTransition, FindConnectingState, ApplyTransitions;
2
3    strategy AddTransition(stateName, guard : string;
4                           prev: object; prevPri : integer)
5    {
6
7      declare pri, minVal, maxVal, avgVal : integer;
8      declare end : object;
9      declare aConnection : object;
10     declare action : string;
11
12     pri:= findAtom("Priority").getIntAttribute("InitialValue");
13
14     if(pri == prevPri + 1) then
15
16       end := self;
17       minVal := findAtom("Min").getIntAttribute("InitialValue");
18       maxVal := findAtom("Max").getIntAttribute("InitialValue");
19       avgVal := (minVal + maxVal) / 2;
20
21       action := stateName;
22       action := action + "=" + intToString(avgVal);
23
24       aConnection := parent().addConnection("Transition", end, prev);
25       aConnection.addAttribute("Guard", guard);
26       aConnection.addAttribute("Action", action);
27
28     endif;
29
30   }
31
32   strategy FindConnectingState(stateName, guard : string)
33   {
34
35     declare pri : integer;
36     declare start : object;
37
38     pri:= findAtom("Priority").getIntAttribute("InitialValue");
39     start := self;
40
41     if(pri < 4) then
42
43       parent().models("State")->
44                      AddTransition(stateName, guard, start, pri);
45
46     endif;
47
48   }
49
50   strategy ApplyTransitions(stateName, guard : string)
51   {
52
53     declare theModel : object;
54
55     theModel := findModel(stateName);
56     theModel.models("State")->FindConnectingState(stateName, guard);
57
58   }
```

Figure 13. Latency Adaptation Transition Strategy

There may be several different variables that can be the focus of QoS adaptation, depending on the contract and goals of an application. In the scenario specified in the strategy of Figure 13, a smaller frame rate is tolerated to maintain a desired latency. The transitions between two sub-states with different priorities will be inserted by the strategies specified in Figure 13. The strategy *Apply-Transitions* in line 50 retrieves the *stateName*, finds out the corresponding sub-state model, and calls another strategy "*FindConnectingState*" (line 32) that will then retrieve the priority of this sub-state. If the current priority is less than 4, another strategy *AddTransition* will be invoked in order to insert the transition from the current sub-state to the next sub-state with a higher priority. Line 12 through Line 14 determine the next sub-state whose priority is just 1 higher than the current state. Line 16 through Line 26 represent the implementation for the insertion of the transition, along with associated attributes.

As a result, the weaving of the strategy from Figure 13 into the model of Figure 10 produces the internal view of the size state, shown in Figure 14. Each state progressively reduces the size of the video frame. The guard condition for the selected transition appears in the lower-right side of the figure. The guard condition states that the transition fires when the latency is not at the desired level, and also when the frame rate has been reduced to its smallest possible size.

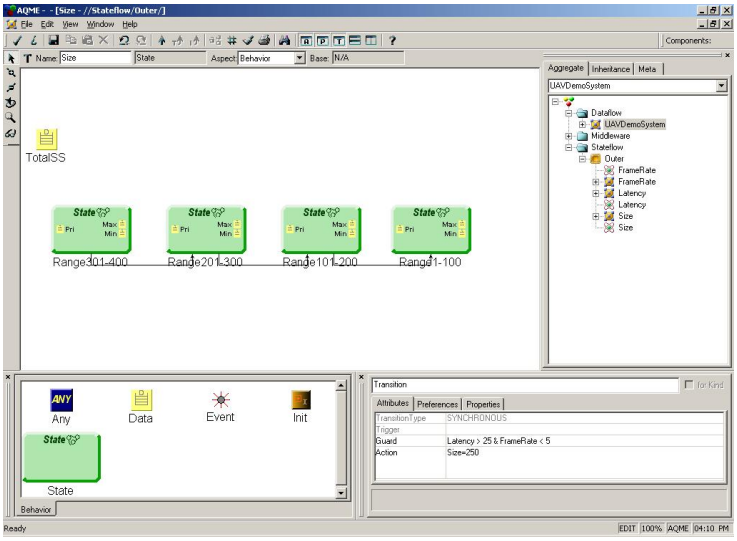


Figure 14. Internal Transitions within the Size State

Related work on modeling aspects for DRE systems propose notations for modeling time using UML (Zhang & Liu, 2005). Likewise, the GenERTiCA tool assists in separating various concerns of a DRE system into aspects within UML and generating code that weaves in the non-functional concerns identified in the models (Wehrmeister et al., 2008). This related work differs from the approach described in this chapter because they are tied to a single modeling language (i.e., UML), as opposed to a more general approach that can be used for multiple domain-specific modeling languages. Moreover, the tool support for weaving at the modeling level itself is not supported in these other works as available in C-SAW.

The Virginia Embedded Systems Toolkit (VEST) (Stankovic et al., 2004)—which is also built as a GME metamodel—is a research effort that has similar goals to the ideas presented in this chapter. VEST supports modeling and analysis of real-time systems and introduces the notion of prescriptive aspects within a design that are programming language independent. A distinction between VEST and the aspect modeling approach presented in this section concerns the generalizabilty of the weaving process. All the modeling aspects possible within VEST can also be specified in C-SAW, but the type of aspect shown in Figure 13 is not possible in VEST. The structure of a prescriptive aspect is limited to the form, "for <some conditional statement on a model property> change <other property>," which is comparatively less powerful than the type of model weaving shown in this chapter.

Scaling a Base Model to Include Multiple UAVs

In addition to the ability to capture crosscutting concerns in models as aspects, C-SAW can assist in transformations that address scalability concerns. In this case study, all the UAVs must share bandwidth and communicate with a ground station that is responsible for allocating bandwidth share to the UAVs according to their operational modes. The behavior of the ground station, as well as its interaction with each UAV, is also modeled in AQML.

It is a relatively modest effort for a user to develop a scenario involving two UAVs that are interacting with one ground station. A challenge arises, however, when the scenario must scale up to production systems containing thousands of UAVs and hundreds of Ground Stations. The models and the synthesis tools available in AQML may assist in mitigating the complexity to some extent by synthesizing a large fraction of the software. A significant share of complexity would be transferred over to model engineers who would need to build models of the larger production scenario.

```
1    defines Start, addUAV_r, addUAV, updateDF;
2
3    aspect Start()
4    {
5      addUAV_r(3,2);
6    }
7
8    strategy addUAV_r(max,idx: integer)
9    {
10     if (idx <= max) then
11       addUAV(idx);
12       addUAV_r(max,idx+1);
13     endif;
14   }
15
16   strategy addUAV(idx: integer)
17   {
18   rootFolder().findFolder("Dataflow").findModel("UAVDemoSystem").updateDF(idx);
19   }
20
21   strategy updateDF(idx:integer)
22   {
23       //The declaration of the local variables are omitted here
24
25       id_str := intToString(idx);
26       uav_csm := rootFolder().findFolder("Stateflow").findModel("UAV_CSM");
27       uav_ins := addInstance("State", "UAV_LC" + id_str, uav_csm);
28
29       tnr := rootFolder().findFolder("Middleware").findModel("TNR");
30       itnr_ins := addInstance("SysCond", "ITNR" + id_str, tnr);
31       otnr_ins := addInstance("SysCond", "OTNR" + id_str, tnr);
32
33       tc := rootFolder().findFolder("Middleware").findModel("TC");
34       tc_ins := addInstance("SysCond", "TC" + id_str, tc);
35
36       sdr := findModel("SDRPipe");
37       sdr_ins := addInstance("Compound", "SDRPipe" + id_str, sdr);
38
39       uins_itnr := uav_ins.findAtom("imag_tnr");
40       itins_tnr := itnr_ins.findAtom("TNR");
41       addConnection("Control", uins_itnr, itins_tnr);
42       addConnection("Control", itins_tnr, uins_itnr);
43
44       // The creations of other connections between the components are omitted
45   }
```

Figure 15. ECL Strategies for Replication of UAVs

Consider the single-UAV model of Figure 6. The difficulty of scaling a model to include additional UAVs is rooted in the implicit relationships between a UAV and the modeled control adaptation. The control logic to perform the requisite adaptation is scattered across numerous entities such that the addition of a new UAV necessitates changes to many other locations to attach the UAV into the model. A model transformation engine like C-SAW applies exceedingly well to this situation. The general solution separates all the complexities of interaction into a strategy that can be reused in many design choices to scale to larger numbers of UAVs.

Figure 15 shows the ECL strategies for inserting two additional UAVs into the modeled system that was originally described in Figure 6. The aspect *Start* (Line 3) is the initiation point of the model weaving process. The strategy *addUAV_r* (Line 8) recursively invokes the strategy *addUAV* (Line 16) that will then call the strategy *updateDF* (Line 21) in order to replicate the UAV and its interactions with the other components in the system. The UAV reference is created from Line 25 to Line 27, which is followed by the creation of the related components such as "*TNR*" (Lines 29 to Line 31), *TC* (Lines 33 and 34), *SDRPipe* (Lines 36 and 37), as well as the connections between the UAV and *TNR* (Line 39 to Line 42). Many other connections are performed in the actual strategy, but they have been omitted here for the sake of brevity (the addition of other interactions follow similarly to those specified here).

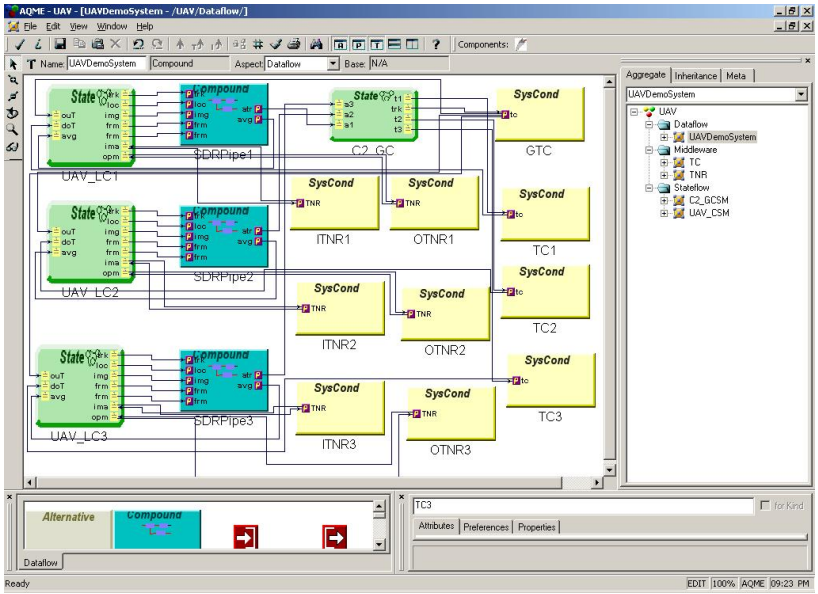


Figure 16. System with 3 UAVs after Weaving

As a result, Figure 16 shows the evolved system with three interacting UAVs. This model can be compared to the original single-UAV model of Figure 6. Without the capability to separate the intent of UAV replication, the ability to scale a model to multiple UAVs presents a manual modeling task that is too burdensome in practice. The use of C-SAW to automate the process permits rapid exploration of a base-line model in order to investigate alternative scenarios involving multiple UAVs. The concept of adding multiple ground stations is not shown here, but would be a similar burden if left solely to manual adaptation of a model. Other examples of scalability of these types of models for several different domains and modeling languages are presented in (Lin et al., 2008).

Synthesis of AQML Models for Simulation and Execution

Using a model-based representation and employing generators to successfully synthesize low-level artifacts from models raises the level of abstraction and provides better tool support for analyzing the adaptation software. Our research has yielded an approach that synthesizes adaptive contract descriptions from models, which permits the creation of larger and more complex contracts than could have

been specified manually. The AQML modeling language has been created in GME to synthesize state machine models into CDL contracts, and also Matlab simulation scripts. Initial results suggest an increase in productivity due to (1) shortening of the design, implement, test, and iterate cycle by providing early simulation, and analysis capabilities and (2) facilitating change maintenance: minimal changes in the weaving of properties into models can make large and consistent changes in the lower level specifications that would not scale well using manual reconfiguration of the models. Details about the generation process from AQML is described in (Neema et al., 2002). The following two sub-sections provide a summary of the artifacts that are generated from AQML.

Matlab Simulation Generator

A primary goal of the modeling approach described in this chapter is to provide integration with tools that can analyze the QoS adaptation from a control-centric viewpoint. A model interpreter for AQML has been created that can translate the QoS adaptation specifications defined in AQML into a Simulink/Stateflow model. Matlab provides an API that is available in the Matlab scripting language (M-file) for procedurally creating and manipulating Simulink/Stateflow models. The simulation generator produces an M-file that uses the API to create Simulink/Stateflow models. Analyses are possible in terms of the time spent in different states, the latency from the time of a change in excitation, to the time of change in outputs in the state machine, and stability of the system. Many details of synthesis to Matlab are documented in (Neema et al., 2002).

Generation of QoS Adaptation Code from AQML

BBN has produced a CDL compiler and a QoS adaptation kernel that can process specifications (i.e., contracts) expressed in CDL. Additional aspect languages have also been created by BBN to support the adaptation effort (Duzan et al., 2004). The CDL compiler translates QoS contracts into artifacts that can execute the adaptation specifications at run-time. The modeling efforts described in this chapter build upon the BBN infrastructure to affect the adaptation specifications captured in the AQML models, i.e., CDL specifications are generated from the AQML models and the BBN tools are used to instantiate these adaptation instructions at run-time in a real system. Neema et al. present results that show the situation where a simple state change within an AQML model translates into dozens of changes that would be needed at the CDL level (Neema et al., 2002), which suggests that QoS adaptation can be modeled with improved scalability when compared to an equivalent process using a textual language.

CONCLUDING REMARKS

Composition of artifacts across the software lifecycle has been the focus of several recent research efforts (Batory, 2006; Gray et al., 2004; Rashid et al., 2003). The synergistic power resulting from a combination of modeling and aspects enables rapid changes to a high-level system specification, which can be synthesized into a simulation or implementation. This chapter described an approach based on MDE for simulating and generating QoS adaptation software for DRE systems using modeling aspects. Our approach focuses on raising the level of abstraction in representing QoS adaptation policies, and providing a control-centric design for the representation and analysis of the adaptation software.

Our approach has been tested and demonstrated on a UAV Video Streaming application described as a case study in this chapter. Although the case study presented a relatively simple scenario, C-SAW enables the modeling of much larger scenarios with a high degree of variability and adaptability. In our experience, the simulation capabilities have been particularly helpful in fine-tuning the adaptation mechanism. In addition to the UAV case study presented in this chapter, the GME and C-SAW have been applied also to Bold Stroke (Roll, 2003), which is a mission avionics computing platform described further in (Gray et al., 2006) in the context of using C-SAW on GME models representing Bold Stroke event channels.

Our investigation into model-driven QoS adaptation of DRE systems has led us to the following observations, which serve as the lessons learned from this contribution:

- *Separation of QoS logic from application source code*: Our experience with QuO and the UAV case study from BBN suggests that separation of QoS adaptation rules into a textual language like CDL offers improvements over QoS adaptation that is otherwise hard-coded directly into the application source code. By separating the QoS adaptation from the source code, it is possible to evolve the QoS adaptation layers more flexibly because the logic relating to each QoS concern can be found in one location, rather than spread across multiple source files.
- *Higher level modeling of QoS concerns*: Although textual languages like CDL can provide improved modularization over source code in a general-purpose programming language (GPL), higher level abstractions associated with domain-specific modeling languages offer even further improvements by assisting developers in specifying QoS concerns and their associated interactions across the components of the underlying application. In several cases, a single modification within the AQML modeling language was found to be equivalent to multiple changes in the corresponding CDL.
- *Adaptation of DRE models through model transformation and aspect weaving*: There are some modeling tasks that are traditionally associated with manual changes to the modeling structure (e.g., concerns like the bandwidth adaptation policy and scalability example from the case study). Such manual changes often require much clicking and typing by the model engineer, such that the modeling task does not scale well (e.g., modifying a single UAV to 10 UAVs would require thousands of mouse clicks); furthermore, issues of correctness emerge when a large number of structural changes are made to a model manually. This chapter presented a contribution toward automated evolution of QoS modeling concerns through model transformation and aspect weaving.
- *Generation of analysis and corresponding artifacts from models:* After the QoS policy has been modeled, it is desirable to simulate the QoS adaptation policy and to analyze various characteristics of the model. Model interpreters, such as those possible within the GME, can generate the necessary files needed to transform the model representation into the format expected by other tools. Furthermore, model interpreters can also translate the model representation into other artifacts, such as CDL and source code in a GPL, for integration within the execution platform of a DRE system. The contribution of this chapter is a justification for treating models as an important artifact for describing QoS adaptation in DRE systems.

We have identified several enhancements as future areas for investigation. We plan to integrate a symbolic model-checking tool, such as SMV (Burch et al., 1992), to enable formal reasoning about the adaptation mechanism. With the aid of this tool, various properties can be established (e.g., liveness, safety, and reachability) about the state machine implementing the adaptation policy. We also plan to strengthen the computation and middleware modeling to facilitate analysis of the application and middleware components. The topic of testing a model transformation rule is another area that will be explored in the future. As model transformations become a large part of the MDE process, it is desirable to have a testing framework that can be used to verify the correctness of each transformation rule. We are currently developing a GME-based testing framework to execute and compare model transformations with C-SAW (Lin et al., 2007; Lin et al., 2005).

A summary of the benefits of concern-driven modeling are shown in Figure 17. A viewpoint is an important concern separation technique that allows different perspectives to be modeled through partitioning mechanisms provided by the GME tool. With viewpoints, attention can be focused on related segments of a model without being overwhelmed by details unrelated to the concern of interest. From the base model, an aspect model weaver can be used to perform global transformations that span the hierarchy of the base model. Aspects at the modeling level enable various design alternatives (e.g.,

properties related to QoS specification) to be explored rapidly. From a transformed model, techniques from generative programming (Czarnecki & Eisenecker, 2000) can be applied to synthesize the model into a form suitable for simulation or model checking. After verification of the models, code can be generated that captures the execution semantics of the modeled system (along with the specified QoS). The iterative nature of the process enables the evaluation of numerous system configurations before actual deployment.

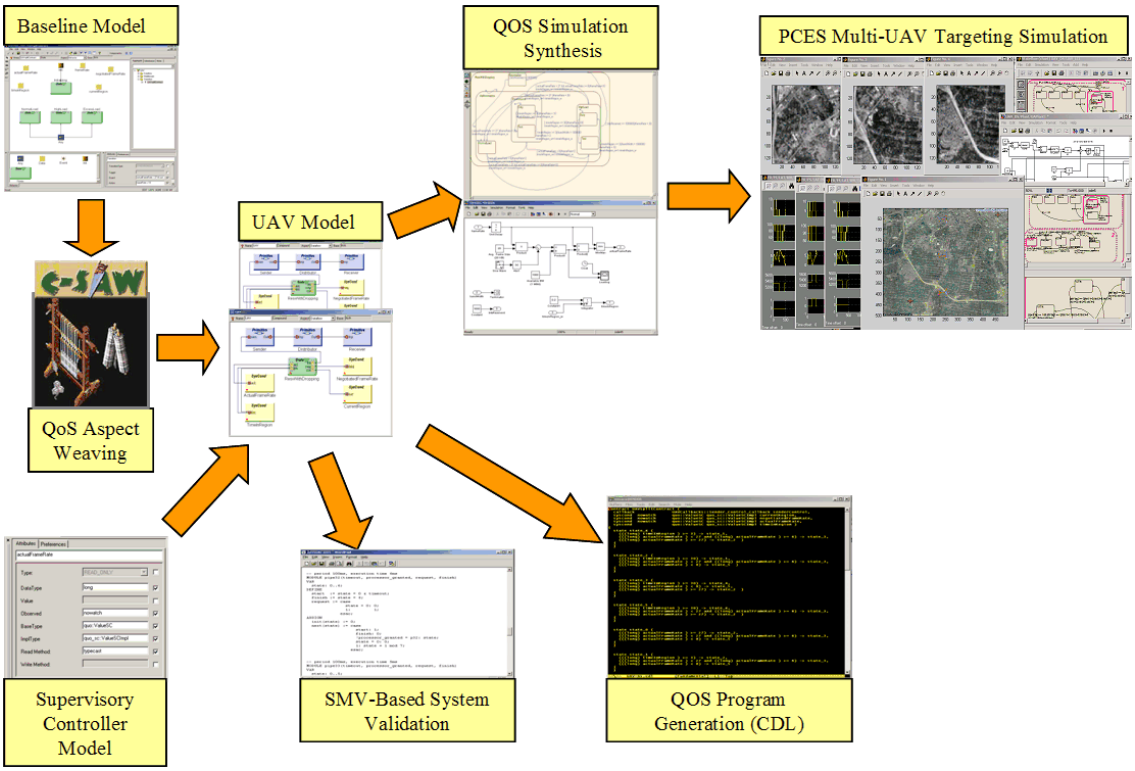


Figure 17. Summary of Concern-Driven QoS Modeling

ACKNOWLEDGMENTS

This work was previously supported by DARPA under the Program Composition of Embedded System (PCES) program and is partially supported by NSF CAREER (CCF-0643725). The authors thank Joseph Loyall and Richard Schantz of BBN Technologies for valuable discussions and comments.

REFERENCES

Balasubramanian, K., Gokhale, A., Karsai, G., Sztipanovits, J., & Neema, S. (2006). Developing Applications Using Model-Driven Design Environments. *IEEE Computer*, 39(2), 33-40.

Batory, D., Sarvela, J.N., & Rauschmayer, A. (2004). Scaling Stepwise Refinement. *IEEE Transactions on Software Engineering*, 30(6), 355-371.

Batory, D. (2006). Multilevel Models in Model-Driven Engineering, Product Lines, and Metaprogramming. *IBM Systems Journal*, 45(3), 527-540.

Bézivin, J., Kurtev, I., Jouault, F., & Valduriez, P. (2006). Model-based DSL Frameworks. *Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, (pp. 602-616). Portland, Oregon.

Dibble, P. (2008). *Real-Time Specification for Java*, Addison-Wesley.

Burch, J., Clarke, E., McMillan, K., Dill, D., & Hwang, L. (1992). Symbolic model checking: 10^20 states and beyond. *Information and Computation*, 98(2), 142-170.

Clarke, S. & Baniassad, E. (2005). *Aspect-Oriented Analysis and Design: The Theme Approach*, Addison-Wesley.

Cottenier, T., van den Berg, A., & Elrad, T. (2007) Motorola WEAVR: Aspect and Model-Driven Engineering. *Journal of Object Technology*, 6(7), 51-88.

Czarnecki, K. & Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley.

Duzan, G., Loyall, J., Schantz, R., Shapiro, R., & Zinky, J. (2004). Building Adaptive Distributed Applications with Middleware and Aspects. *International Conference on Aspect-Oriented Software Development (AOSD)*, (pp. 66-73), Lancaster, UK.

Elrad, T., Aldawud, A., & Bader, A., (2002). Aspect-Oriented Modeling: Bridging the Gap between Modeling and Design. *Generative Programming and Component Engineering (GPCE)*, (pp. 189-201), Pittsburgh, Pennsylvania.

Filman, R., & Friedman, D., (2000). Aspect-Oriented Programming is Quantification and Obliviousness. *OOPSLA Workshop on Advanced Separation of Concerns*, Minneapolis, Minnesota.

Gokhale, A., Balasubramanian, K., Krishna, A., Balasubramanian, J., Edwards, G., Deng, G., Turkay, E., Parsons, J., & Schmidt, D. (2008). Model-driven Middleware: A New Paradigm for Developing Distributed Real-time and Embedded Systems. *Science of Computer Programming*, 73(1), 39-58.

Gray, J., Bapty, T., Neema, S., & Tuck, J. (2001). Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, 44(10), 87-93.

Gray, J., Sztipanovits, J., Schmidt, D., Bapty, T., Neema, S., & Gokhale, A. (2004). Two-level Aspect Weaving to Support Evolution of Model-Driven Synthesis. In *Aspect-Oriented Software Development*, (pp. 681-710), Addison-Wesley.

Gray, J., Lin, Y., & Zhang, J. (2006). Automating Change Evolution in Model-Driven Engineering. *IEEE Computer*, 39(2), 41-48.

Gray, J., Tolvanen, J-P., Kelly, S., Gokhale, A., Neema, S., & Sprinkle, J. (2007). Domain-Specific Modeling. In *Handbook on Dynamic System Modeling,* (pp. 7-1 – 7-20), CRC Press.

Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3), 231-274.

Karr, D., Rodrigues, C., Loyall, J., Schantz, R., Krishnamurthy, Y., Pyarali, I., & Schmidt, D. (2001). Application of the QuO Quality-of-Service Framework to a Distributed Video Application. *International Symposium on Distributed Objects and Applications*, (pp. 299-309), Rome, Italy.

Karsai, G., Maroti, M., Lédeczi, A., Gray, J., & Sztipanovits, J. (2004). Composition and Cloning in Modeling and Metamodeling Languages. *IEEE Transactions. on Control System Technology*, 12(2), 263-278.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., & Irwin, J. (1997). Aspect-Oriented Programming. *European Conference on Object-Oriented Programming (ECOOP)*, (pp. 220-242), Jyväskylä, Finland.

Lédeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., & Karsai, G. (2001). Composing Domain-Specific Design Environments. *IEEE Computer*, 34(11), 44-51.

Lin, Y., Zhang, J., & Gray, J. (2005). A Framework for Testing Model Transformations. In *Model-Driven Software Development*, (pp. 219-236), Springer.

Lin, Y. (2007). *A Model Transformation Approach to Automated Model Evolution*, Doctoral Dissertation, University of Alabama at Birmingham, Department of Computer and Information Sciences, available at http://www.cis.uab.edu/softcom/dissertations/LinYuehua.pdf

Lin, Y., Gray, J., & Jouault, F. (2007). DSMDiff: A Differentiation Tool for Domain-Specific Models. *European Journal of Information Systems*, 16(4), 349-361.

Lin, Y., Gray, J., Zhang, J., Nordstrom, S., Gokhale, A., Neema, S., & Gokhale, S. (2008). Model Replication: Transformations to Address Model Scalability. *Software: Practice and Experience*, 38(14), 1475-1497.

Loyall, J., Schantz, R., Zinky, J., Pal, P., Shapiro, R., Rodrigues, C., Atighetchi, M., Karr, D., Gossett, J., & Gill, C. (2001). Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications. *IEEE International Conference on Distributed Computing Systems (ICDCS)*, (pp. 625-634), Phoenix, Arizona.

Mernik, M., Heering, J., & Sloane, A. (2005). When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4), 316-344.

Neema, S., Bapty, T., Gray, G., & Gokhale, A. (2002). Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems. *Generative Programming and Component Engineering (GPCE)*, (pp. 236-251), Pittsburgh, Pennsylvania.

Neema, S., Sztipanovits, J., Karsai, G., & Butts, K. (2003). Constraint-Based Design-Space Exploration and Model Synthesis. *International Conference on Embedded Software (EMSOFT)*, (pp. 290-305), Philadelphia, Pennsylvania.

Nuseibeh, B., Kramer, J., & Finkelstein, A. (1994). A Framework for Expressing the Relationship Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering,* 20(10), 760-773.

Rashid, A., Moreira, A., & Araújo, J. (2003). Modularization and Composition of Aspectual Requirements. *International Conference on Aspect-Oriented Software Development (AOSD)*, (pp. 11-20), Boston, Massachusetts.

Reddy, Y., Ghosh, S., France, R., Straw, G., Bieman, J., McEachen, N., Song, E., & Georg, G. (2006). Directives for Composing Aspect-Oriented Design Class Models. *Transactions on Aspect-Oriented Software Development*, 1(1), 75-105.

Roll, W. (2003). Towards Model-Based and CCM-Based Applications for Real-Time Systems," *International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, (pp. 75-82), Hokkaido, Japan.

Schantz, R., & Schmidt, D. (2001). Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications. In *Encyclopedia of Software Engineering*, John Wiley and Sons.

Schantz, R., Loyall, J., Atighetchi, M., & Pal, P. (2002). Packaging Quality of Service Control Behaviors for Reuse. *International Symposium on Object-oriented Real-time Distributed Computing (ISORC)*, (pp. 375-385), Washington, DC.

Schmidt, D. (2006). Model-Driven Engineering. *IEEE Computer*, 39(2), 25-32.

Sharma, P., Loyall, J., Heineman, G., Schantz, R., Shapiro, R., & Duzan, G. (2004). Component-Based Dynamic QoS Adaptations in Distributed Real-Time and Embedded Systems. *International Symposium on Distributed Objects and Applications (DOA)*, (pp. 1208-1224), Agia Napa, Cyprus.

Stankovic, J., Nagaraddi, P., Yu, Z., He, Z. & Ellis, B. (2004) Exploiting Prescriptive Aspects: A Design Time Capability, *International Conference on Embedded Software (EMSOFT)*, (pp. 165-174), Pisa, Italy.

Stein, D., Hanenberg, S., & Unland, R. (2006). Expressing Different Conceptual Models of Join Point Selections in Aspect-Oriented Design. *International Conference on Aspect-Oriented Software Development (AOSD)*, (pp. 15-26), Bonn, Germany.

Sztipanovits, J., Karsai, G. (1997). Model-Integrated Computing. *IEEE Computer*, 30(4), 10-12.

Tarr, P. (2003). Toward a More Piece-ful World. *Generative Programming and Component Engineering (GPCE),* (pp. 265-266), Erfurt, Germany.

Tarr, P., Ossher, H., Harrison, W., & Sutton, S. (1999). N Degrees of Separation: Multi-Dimensional Separation of Concerns. *International Conference on Software Engineering (ICSE)*, (pp. 107-119), Los Angeles, California.

Wehrmeister, M., Freitas, E., Pereira, C., & Rammig, F. (2008). GenERTiCA: A Tool for Code Generation and Aspects Weaving. *International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, (pp. 234 – 238), Orlando, Florida.

Zhang, L., & Liu, R. (2005). Aspect-oriented Real-time System Modeling Method based on UML. *International Conference on Embedded and Real-Time Computing Systems and Applications (RTAS)*, (pp. 373-376), San Francisco, California.