

Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems

Sandeep Neema, Ted Bapty, Jeff Gray, Aniruddha Gokhale
Institute for Software Integrated Systems,
Vanderbilt University, Nashville
{sandeep|bapty|jgray|gokhale}@isis-server.vuse.vanderbilt.edu

Abstract.

This paper presents a model-driven approach for generating quality-of-service (QoS) adaptation in Distributed Real-Time Embedded (DRE) Systems. The approach involves the creation of high-level graphical models representing the QoS adaptation policies. The models are constructed using a domain-specific modeling language - the Adaptive Quality Modeling Language (AQML). Multiple generators have been developed using the Model-Integrated Computing (MIC) framework to create low-level artifacts for simulation and implementation of the adaptation policies that are captured in the models. A simulation generator tool synthesizes artifacts for Matlab® Simulink®/Stateflow® (a popular commercial tool), providing the ability to simulate and analyze the QoS adaptation policy. An implementation generator creates artifacts for Quality Objects (QuO), a QoS adaptation software infrastructure developed at BBN, for execution of QoS adaptation in DRE systems. A case study in applying this approach to an Unmanned Aerial Vehicle – Video Streaming application is presented. This approach has goals that are similar to those specified in the OMG's Model-Driven Architecture initiative.

1. Introduction

Rapid advances in middleware technologies have given rise to a new generation of highly complex, object-oriented Distributed Real-Time Embedded (DRE) systems based on technologies such as RT-CORBA, COM+, RMI, among others. As observed in [Schantz and Schmidt, 01], middleware solutions promote software reuse resulting in better productivity. However, within the context of DRE systems, specifying and satisfying quality-of-service (QoS) requirements demand ad-hoc, problem-specific code optimizations. Such modifications go against the very principles of reuse and portability.

There have been some attempts to improve this situation by introducing a programmable QoS adaptation layer on top of the middleware infrastructure. The key idea behind the adaptation layer is to offer *separation of concerns* with respect to QoS requirements. Such separation provides improved modularization for separating QoS

requirements from the functional parts of the software. Quality Objects (QuO), developed by BBN, is one such adaptation layer [Loyall et al., 01]. QuO provides an extension to the Object Management Group's (OMG) Interface Definition Language (IDL). This extension, known as the Contract Definition Language (CDL), supports the specification of QoS requirements and adaptation policies. Contracts written in CDL are compiled to create stubs that monitor and adapt the QoS parameters when the system is operational. These stubs are integrated into the QuO kernel.

From a software engineering standpoint this approach works well, however, there are a few drawbacks to using CDL alone:

1. The control-centric nature of the QoS adaptation extends beyond the software. QoS parameters in a DRE system have a significant impact on the dynamics of the overall physical system. Owing to the complex and non-linear dynamics, it is very difficult 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. Therefore, sophisticated tools are needed to design, simulate, and analyze the QoS adaptation software from a controls perspective.
2. For scalability reasons, the CDL offers a much lower level of abstraction (textual code-based). Even for a moderately large system, the CDL specifications can grow quite large. Instrumenting a small change to the adaptation policy requires making several changes manually, while ensuring that all these changes are consistent in the CDL specification.

In this paper, we present an approach based on the principles of Model-Integrated Computing (MIC) [Lédeczi et al., 01], which addresses these issues. Our approach involves creating a graphical modeling environment that allows a DRE system designer to graphically model the DRE system and the QoS adaptation policy using standardized notations, such as Statecharts [Harel, 87] and Dataflow. A generator tool synthesizes artifacts for Matlab® Simulink®/Stateflow® (a popular commercial tool) providing the ability to simulate and analyze the QoS adaptation policy. This gives significant assurance that the system will perform as desired. A second generator tool creates CDL specifications from the QoS adaptation models. The generated CDL is then compiled into executable artifacts. The approach described in this paper has goals that are similar to those specified in the OMG's Model-Driven Architecture (MDA) [Bézivin, 01].

The rest of this paper is organized as follows. Section 2 presents the modeling environment and the modeling concepts required by our approach. Section 3 introduces the generator that creates simulation artifacts from the models. Section 4 describes another generator that creates CDL specifications from the models. A short case study in applying the approach to a video streaming application, within the context of an Unmanned Aerial Vehicle (UAV), is presented in section 5. The paper ends with some concluding remarks in section 6.

2. Modeling Paradigm

The MIC infrastructure provides a unified software architecture and framework for creating a Model-Integrated Program Synthesis (MIPS) environment [Lédeczi et al., 01]. The core components of the MIC 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. The new environment is domain-specific and 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 the MIC technology, the modeling concepts to be instantiated in the MIPS environment are specified in a *meta-modeling* language [Nordstrom et al., 99]. A *meta-model* 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 inter-relationships between different objects. The meta-modeling language also provides for the specification of visual presentation of the objects in the graphical model editor [Nordstrom et al., 99].

The Adaptive Quality Modeling Language (AQML) presented here models the following key aspects of a DRE system:

1. **QoS Adaptation Modeling** – In this first category, the adaptation of QoS properties of the DRE system is modeled. The designer 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 updated QoS information, and the events that trigger the transitions. These properties are modeled using an extended Finite-State Machine (FSM) modeling formalism [Harel, 87].
2. **Computation Modeling** – In this category, the computational aspect of a DRE system is modeled. A dataflow model is created in order to specify the various computational components and their interaction. An extended dataflow modeling formalism is employed.
3. **Middleware Modeling** – In this category, the middleware services, the system monitors, and the tunable “knobs” (i.e., the parameters being provided by the middleware) are modeled.

The metamodel of each of these modeling categories and their interaction in the AQML is described below.

QoS Adaptation Modeling

Stateflow models capture the QoS adaptive behavior of the system. A *Discrete Finite State Machine* representation, extended with hierarchy and concurrency, is selected

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 1 illustrates the QoS adaptation aspect of the AQML.

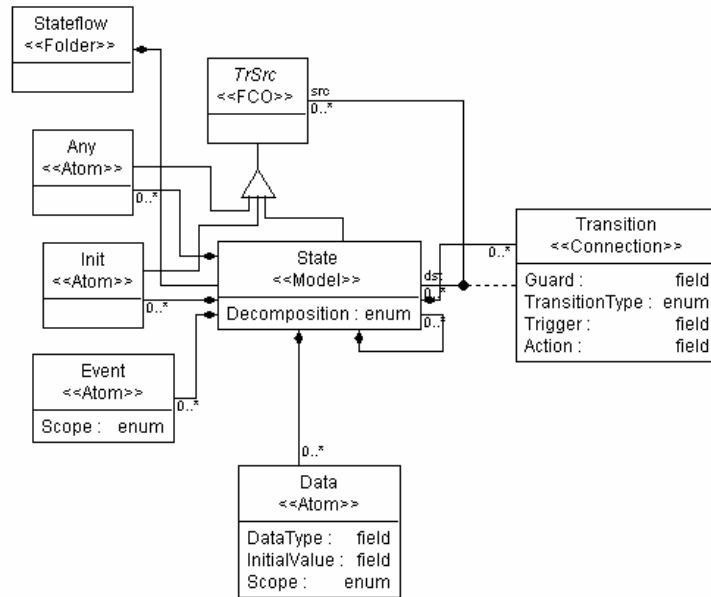


Fig. 1. Metamodel of QoS Adaptation Modeling

The main concept in a finite state machine representation is a *state*. States define 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 the MIC framework.

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 MIC framework. 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 modeling paradigm, *Event* objects capture the Boolean event variables, and the *Data* objects capture arbitrary data vari-

ables. 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 modeling category 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 2 illustrates the computation aspect of the AQML. The different objects and their inter-relationships are described below.

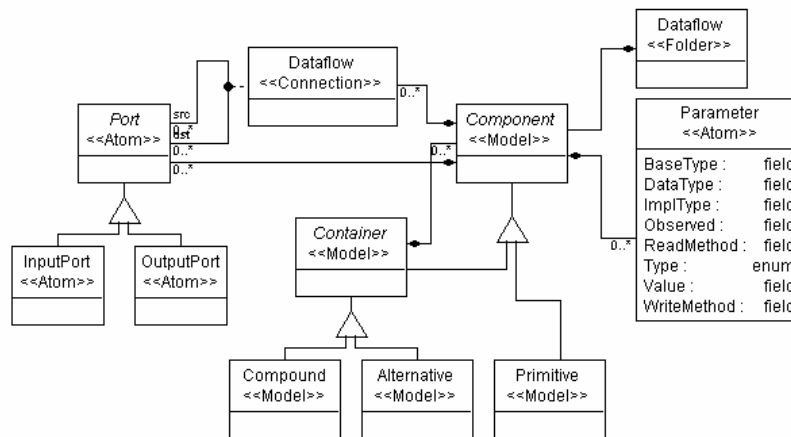


Fig. 2. Metamodel of Computation Architecture Modeling

The computational structure is modeled with the following classes of objects: *Compounds* 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 class 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 important concept relevant to QoS adaptive DRE systems is the notion of parameters. Parameters are the tunable “knobs” that are used by the adaptation mecha-

nism to tailor the behavior of the components such that desired QoS properties are maintained. Parameters can be contained in Compounds and Primitives. 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 category, the components of the middleware are modeled. These components include the services and the system conditions provided by the middleware. Examples of services include an Audio-Video Streaming service, Bandwidth reservation service, Timing service, and Event service, among others. System conditions are components that provide quantitative diagnostic information about the middleware. Examples of these include observed throughput, bandwidth, latencies, and frame-rates. Figure 3 illustrates the middleware modeling aspect of the AQML.

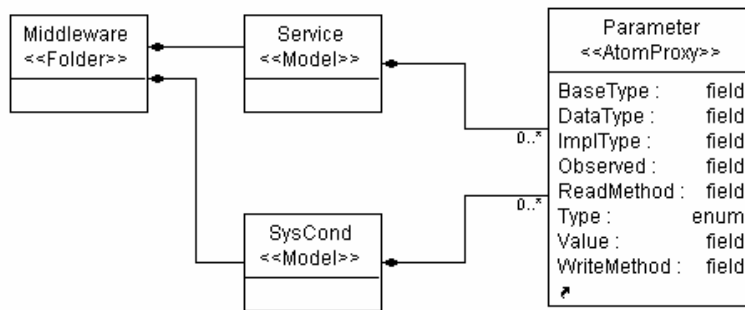


Fig. 3. Metamodel of Middleware Modeling

The *Service* object represents the services provided by the middleware. Services can contain parameters that are the tunable knobs 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.

Observe that we do not facilitate a detailed modeling of the middleware components or the dataflow components. This is 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

In this category, the interaction of the previous three modeling categories (illustrated in Figure 4) is specified. As described earlier, the Data/Event objects within the

Stateflow model form the interface of the state machine. Within the Computation and the Middleware models, Parameters form the control interfaces. The interaction of the QoS adaptation (captured in Stateflow models), and the middleware and application (modeled in the Middleware and 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 or a Computation model.

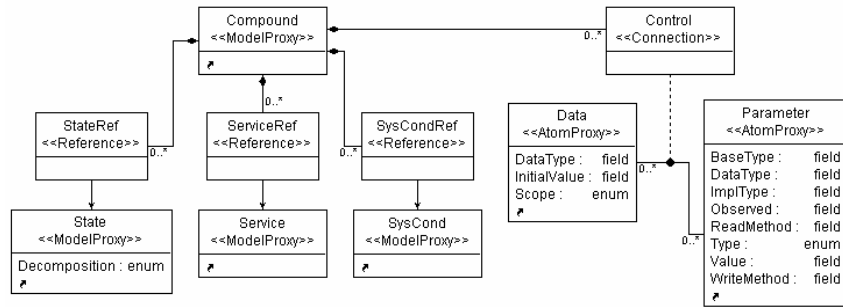


Fig. 4. Metamodel of interaction of QoS adaptation with Middleware and Computation modeling

In the MIC framework, connections between objects that are not contained within the same model are not supported. Therefore, we create references, which are equivalent to a “reference” or a “pointer” in a programming language. 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.

4. Simulation Generator

One of the primary goals of our approach, as identified earlier, is to be able to provide integration with tools that can analyze the QoS adaptation from a control-centric viewpoint. Matlab® Simulink®/Stateflow® is an extremely popular commercial tool that is routinely used by control engineers to design and simulate discrete controllers. Simulink provides the ability to model hybrid systems (mixed continuous and discrete dynamics) in a block-diagrammatic notation. Stateflow provides the ability to model hierarchical parallel state machines in a Statechart like notation [Harel, 87]. A Stateflow model can be inserted in a Simulink model as a block and the Simulink blocks can provide input stimulus and receive outputs from the Stateflow block. The Simulink/Stateflow model can be simulated within the Matlab framework for a desired time period, which in effect steps through the state-machine for the given input excitation. The responses from the state machine can be graphically plotted and the trajectory of the state machine can be visually observed, as well as recorded, for post-

analysis. Additional 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, stability of the system, etc. Thus, the Matlab Simulink/Stateflow tool suite provides an extremely convenient and intuitive framework for observing and verifying the behavior of the system.

We have implemented a generator, using our model interpretation technology, that can translate the QoS adaptation specifications (modeled using the 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.

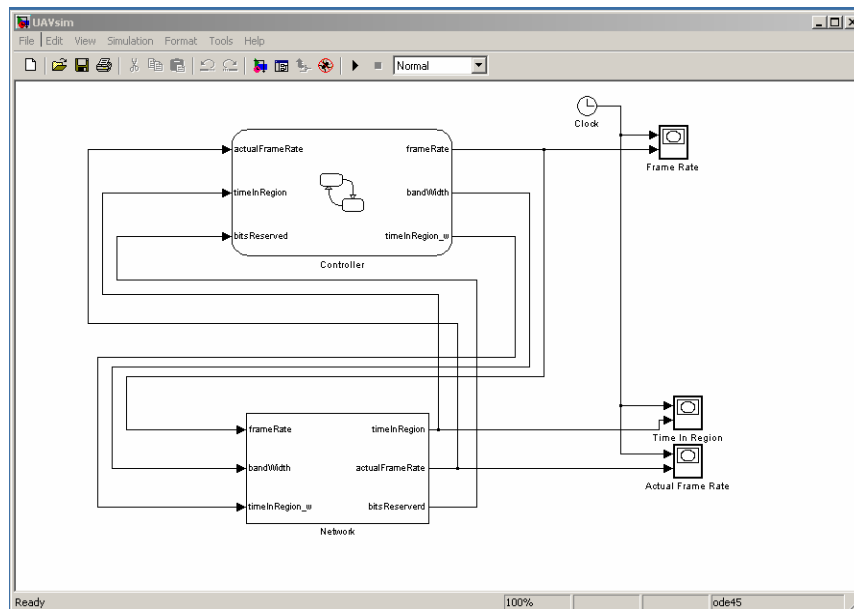


Fig. 5. Top-level structure of the generated Simulink/Stateflow Model

The structure of the generated Simulink/Stateflow model is shown in Figure 5. There are two main blocks in the generated Simulink model. A Stateflow block is generated that represents the QoS adaptation engine (labeled Controller in Figure 5). A Subsystem block is generated that represents the simulated middleware and the application (labeled Network in Figure 5). Only the interface of this block is generated. The user models the dynamics of the physical network, the middleware, and the application composite, as mathematical equations created using Simulink blocks within this Subsystem. The fidelity of this model is dependent upon the user. A variety of models may be created, ranging from coarse and simple, to highly fine-grained and extremely complex. It must be noted here that these models are dependent on the underlying physical network, and not dependent on the controller itself. Thus, for a new

network, a user may have to create a new model; however, while simulating and verifying the modeled controller, there is no need to recreate the network simulation model every time there is a change in the controller.

The main work of the simulation generation algorithm is in synthesizing the Stateflow block. The Stateflow representation within Matlab has almost a one-to-one equivalence with the Stateflow representation within AQML. However, the representation of hierarchy is somewhat complicated in Matlab Stateflow. Although the AQML hierarchy is established by an actual containment relationship over objects, the Matlab hierarchy is enforced strictly through graphical containment. Thus, all the rectangles representing child states are graphically bounded by the rectangle symbolizing the parent state. Any overlap is flagged as a syntax error by the Matlab Stateflow simulator.

The Stateflow block generation is essentially a two-pass interpretation. The first-pass calculates the bounding rectangles for each state according to the containment relationships. The second-pass emits the API calls in the M-file to create states with appropriate bounding rectangles. The second-pass also emits API calls for creating data variables, event variables, and transitions.

An example of the generated M-file and the Simulink/Stateflow diagram created upon executing the script is shown in the case study.

5. CDL Generator

Research at BBN on DRE systems and QoS adaptation resulted in the CDL – a domain-specific language (based on OMG IDL) for specifying QoS contracts. This research also produced a CDL compiler and a QoS adaptation kernel that can process specifications (contracts) expressed in CDL. The CDL compiler translates QoS contracts into artifacts that can execute the adaptation specifications at run-time. CDL is a textual language, and it has a state-machine like flavor (see [Loyall et al., 01] for details). Our research efforts build upon their work, and we utilize their infrastructure to affect the adaptation specifications captured in the AQML models. That is, we generate CDL specifications from the AQML models and use the BBN QuO tools to actually instantiate these adaptation instructions at runtime.

Although the CDL has a state-machine like flavor and can implement hierarchical finite state machines, it does not support the notion of parallelism in the state machine description. To compensate, the AQML CDL generator procedurally explores the state-space captured by the hierarchical, parallel, finite-state machines in the AQML, and translates it into flat finite state machines. A simple algorithm to procedurally explore the state-space captured by a hierarchical, parallel, finite state machine could be exponential in complexity. Instead, we use an approach based on symbolic methods to explore the state-space. We use a variant of a prefix-based encoding scheme to assign an encoding (a string of Boolean values) to each state in the state machine (see [Neema, 01] for details). Given this encoding, each state can be represented as a Boolean formula, and the complete state-space can be composed symbolically using these Boolean formulae. The satisfying valuations of the Boolean formula represent the entire state-space corresponding to a state in the flattened state machine representation.

Transitions in the flat finite-state machine representation can be determined procedurally by determining the constituent components of a satisfying valuation from the encoding.

A second challenge in translating the AQML models to CDL is in the synthesis of the transition triggers, guards, and actions. In AQML the triggers, guards, and actions, are written using a standard expression language (from Statecharts). Further, the expressions involve the data and event variables declared within the state machine description. In the CDL, however, the expressions are written in an IDL/C++ flavor. The expressions involve method calls over the SysCond objects, which are passed as arguments to the contract. Therefore, the CDL generator parses the trigger, guard, and action expressions. The abstract syntax tree created by the parser is traversed by the generator. During traversal, when data or event variables are referenced in the expression, the generator determines the middleware or application parameter that the data variable is connected to and emits the equivalent of a *get* or *set* method call on the parameter object (depending on the context of the reference). The exact name of the method is available to the generator via the ReadMethod and WriteMethod attributes of the Parameter object in the Middleware and Computation Model (see Figure 2).

The CDL generator performs a multi-pass interpretation. The first pass parses the trigger, guard, and action expressions of the transitions, and builds an individual abstract syntax tree. A second pass creates a flat finite-state machine representation from the hierarchical, parallel, finite-state machine representation. The final pass traverses the flat finite-state machine representation, evaluates the abstract syntax tree created for trigger, guard, and action expressions of the transitions, and emits CDL specifications that represent an equivalent flat finite-state machine representation.

The CDL generation presented above is further illustrated with an example in the following section.

6. Case Study

The case study presented in this section is based upon a video streaming application for an Unmanned Aerial Vehicle (UAV) [Karr et al., 01]. There are several things that make this a complex and challenging problem (i.e., the real-time requirements, resource constraints, and the distributed nature). Figure 6 shows a UAV video application scenario. This is a navy application where a number of UAVs are simultaneously transmitting surveillance video from different regions in a battlefield. A distributor node on a shipboard receives these video streams and sends it to different receivers on the ship that are interested in monitoring those video streams and responsible for making tactical decisions about deployment and guidance. Some of these receivers may be performing automatic target recognition while guiding other Unmanned Combat Aerial Vehicles (UCAV) for weapon launching. There are a few interesting observations to make: 1) the link between the UAV to the ship 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, nature of threat, etc; 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 (processors, networks, switches) involved in the entire application; and 5) the scenario is highly dynamic (i.e., UAVs frequently enter and leave the battle field). Given these complex requirements, a QoS-enabled middleware solution has been proposed for this application [Karr et al., 01], [Loyall et al., 01]. Due to the highly dynamic nature of the application scenario, the adaptation of the QoS properties is mandatory.

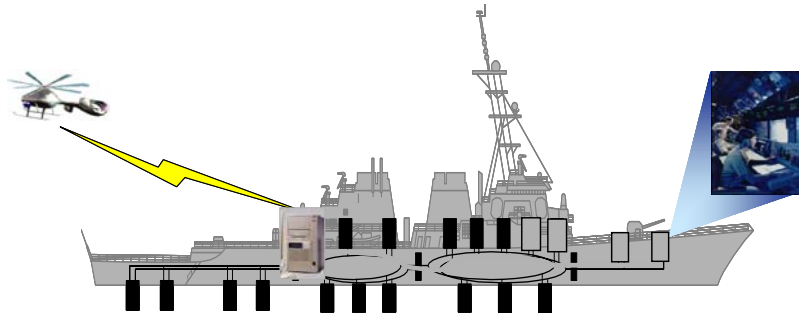


Fig. 6. BBN UAV Video Streaming Application Scenario (reprinted from [Karr et al.,01], with permission from BBN)

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. However, when the communication and computational resources are loaded, 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 of reducing the transmission rate: a) reduce the frame rate by dropping frames, b) reduce the image quality per frame, or c) reduce the frame size. Depending on the actual scenario, one or more of these situations may apply. In the example of this section, we consider dropping the frame rate only.

Figure 7 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, some other data-variables may enact the adaptation actions that are being performed in the transitions. Notice that the attribute window at the bottom-right corner of the figure 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 DRE 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. Figure 8 shows an exploration into the hierarchy of the

HighLoad state. A Duty and Test state are being depicted in this figure. The general idea is that when the system enters a HighLoad state, the frame rate is reduced. However, the adaptation periodically probes the network by entering a Test state, which bumps up the frame rate. If the transient load has disappeared, then the actualFrameRate variable goes up (indicating that the network can sustain the desired frame rate). When this happens, the system switches into the NormalLoad state; otherwise, it goes back into the Duty state of the HighLoad state.

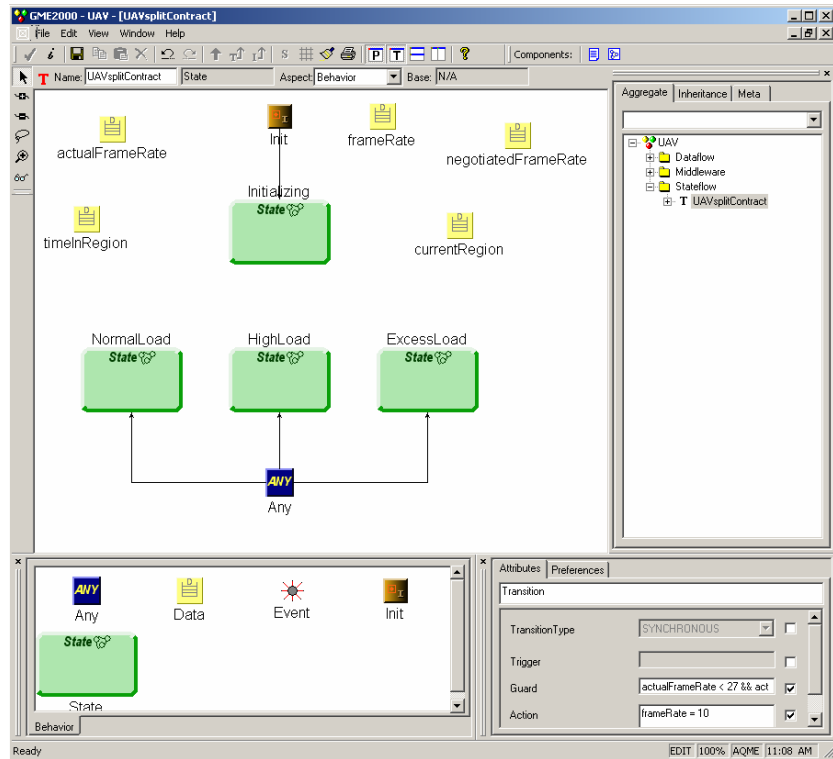


Fig. 7. Model of QoS adaptation in AQML

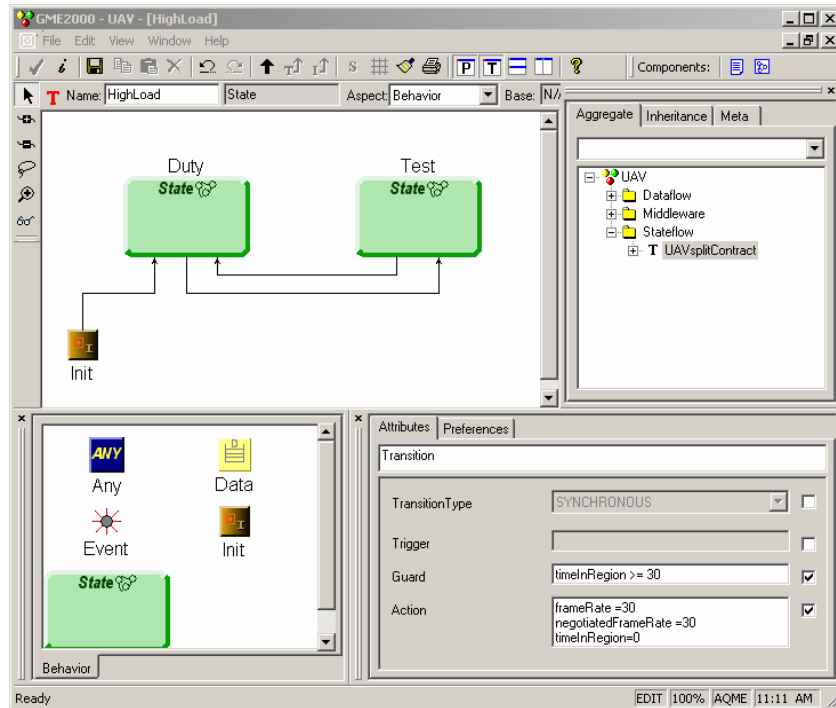


Fig. 8. The HighLoad State Model

Figure 9 shows an M-file that was generated from the model in Figure 7. Figure 10 shows the generated Stateflow model, which in effect is equivalent to the state machine captured in the AQML models. Figure 11 illustrates a network simulation model that is created by hand. It is a fairly simple mathematical model of the network, and it artificially loads the network with a periodic sinusoidal load (modeled by the sine-wave block in Simulink). Figure 12 and Figure 13 presents the results of the simulation. The behavior of the adaptive system can be intuitively observed and understood from these plots. It can be seen that when the load crosses a threshold, the adaptation engine causes the frame rate to reduce adaptively (thus minimizing the latency). When the load vanishes, the frame rate is enhanced again. Some important information about the adaptation (in terms of lead times, response times, and stability) can be gathered from these plots. If the response is too quick, then the network may enter into oscillations. The network may also get into oscillatory behavior if the frame-rate reduction step is too high. On the other hand, if the frame-rate reduction step is too low, the duration for which the latency of the network stays high may be very large.

```
d:\projects\pces\vss\BBN\Models\UAVsim.m
File Edit View Text Debug Breakpoints Web Window Help
Stack: Base
1 % Matlab script for creation of a Stateflow Model
2 % generated: Tuesday, December 25, 2001
3
4 sfnew('UAV');
5 root = sfroot;
6 mach = root.find('-isa', 'Stateflow.Machine');
7 chart = mach.find('-isa', 'Stateflow.Chart');
8 chart.Name = 'Controller';
9 set_param(gcbh,'Position',[120 40 300 160]);
10
11 % create states
12 s0 = Stateflow.State(chart);
13 s0.Name = 'ResvWithDropping';
14 s0.LabelString = 'ResvWithDropping';
15 s0.Decomposition = 'EXCLUSIVE_OR';
16 s0.Position = [0 0 490 440];
17 s1 = Stateflow.State(chart);
18 s1.Name = 'NoReservation';
19 s1.LabelString = 'NoReservation';
20 s1.Decomposition = 'EXCLUSIVE_OR';
21 s1.Position = [20 40 450 280];
22 s2 = Stateflow.State(chart);
23 s2.Name = 'ExcessLoad';
24 s2.LabelString = 'ExcessLoad';
25 s2.Decomposition = 'EXCLUSIVE_OR';
26 s2.Position = [40 80 130 220];
27 s3 = Stateflow.State(chart);
28 s3.Name = 'Test';
29 s3.LabelString = 'Test';
30 s3.Decomposition = 'EXCLUSIVE_OR';
31 s3.Position = [60 120 90 60];
32 s4 = Stateflow.State(chart);
33 s4.Name = 'Duty';
34 s4.LabelString = 'Duty';
35 s4.Decomposition = 'EXCLUSIVE_OR';
36 s4.Position = [60 200 90 60];
37 s5 = Stateflow.State(chart);
38 s5.Name = 'HighLoad';
39 s5.LabelString = 'HighLoad';
40 s5.Decomposition = 'EXCLUSIVE_OR';
Ready
```

Fig. 9. Matlab M-file generated from UAV model

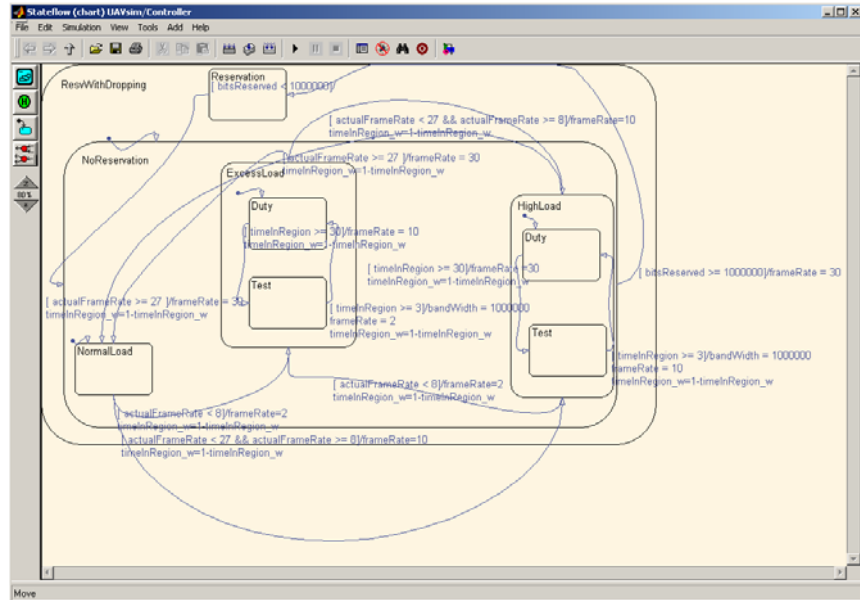


Fig. 10. Generated Matlab Stateflow model

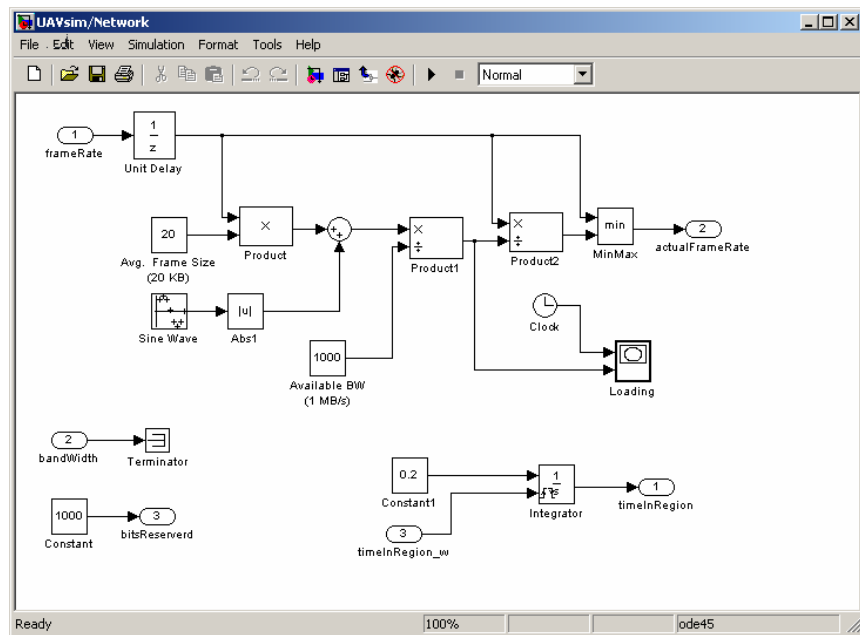


Fig. 11. Simulation model of the network

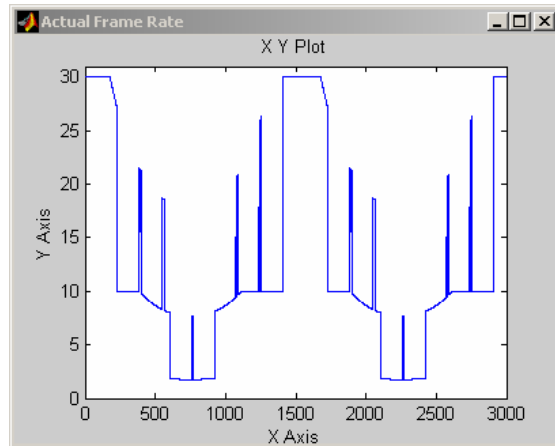


Fig. 12. Simulation results – Actual Frame Rate

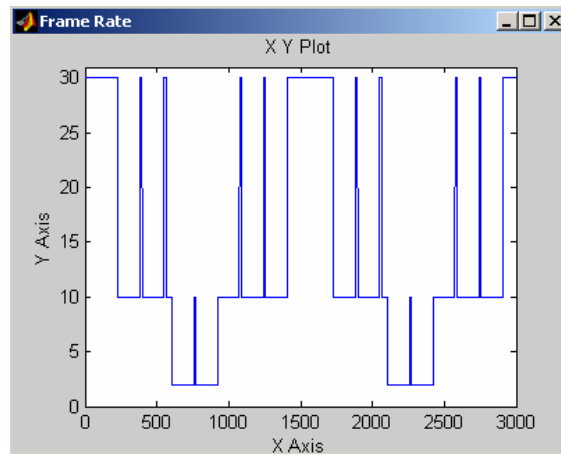


Fig. 13. Simulation results – Desired Frame Rate

Figure 14 illustrates a CDL file that was generated from the same model. This CDL file is processed by the QuoGen compiler to generate runtime artifacts for the Quo kernel.

7. Conclusions

This paper presented an approach based on Model-Integrated Computing for simulating and generating QoS adaptation software for Distributed Real-time Embedded systems. The key focus of the approach is on raising the level of abstraction in represent-

ing QoS adaptation policies, and providing a control-centric design for the representation and analysis of the adaptation software. Using a model-based representation and employing generators to create low-level artifacts from the models, we have been successful in raising the level of abstraction, and providing better tool support for analyzing the adaptation software. At the same time, our approach results in increased productivity as we 1) shorten the design, test, and iterate cycle by providing early simulation, and analysis capabilities, and 2) facilitate change maintenance as minimal changes in the models can make large and consistent changes in the low-level (CDL) specifications. This approach has similar goals to the OMG's Model-Driven Architecture (MDA) initiative.

```

contract UAVsplitContract (
  callback UAVcallbacks::Sender_Control_Callback senderControl,
  syscond nowatch quo::valueSC quo_sc::valueSCImpl currentRegion,
  syscond nowatch quo::valueSC quo_sc::valueSCImpl negotiatedFrameRate,
  syscond nowatch quo::valueSC quo_sc::valueSCImpl actualFrameRate,
  syscond quo::valueSC quo_sc::valueSCImpl timeInRegion )
{
  state state_6 (
    (((long) timeInRegion ) >= 3) -> state_5,
    (((long) actualFrameRate ) < 27 and ((long) actualFrameRate ) >= 8) -> state_3,
    (((long) actualFrameRate ) >= 27) -> state_2 )
  {
  }

  state state_2 (
    (((long) timeInRegion ) >= 30) -> state_1,
    (((long) actualFrameRate ) < 27 and ((long) actualFrameRate ) >= 8) -> state_3,
    (((long) actualFrameRate ) < 8) -> state_5 )
  {
  }

  state state_3 (
    (((long) timeInRegion ) >= 30) -> state_4,
    (((long) actualFrameRate ) < 8) -> state_5,
    (((long) actualFrameRate ) >= 27) -> state_2 )
  {
  }

  state state_5 (
    (((long) timeInRegion ) >= 30) -> state_6,
    (((long) actualFrameRate ) < 27 and ((long) actualFrameRate ) >= 8) -> state_3,
    (((long) actualFrameRate ) >= 27) -> state_2 )
  {
  }

  state state_0 (
    (((long) actualFrameRate ) >= 27) -> state_2,
    (((long) actualFrameRate ) < 27 and ((long) actualFrameRate ) >= 8) -> state_3,
    (((long) actualFrameRate ) < 8) -> state_5 )
  {
  }

  state state_1 (
    (((long) timeInRegion ) >= 3) -> state_2,
    (((long) actualFrameRate ) < 27 and ((long) actualFrameRate ) >= 8) -> state_3,
    (((long) actualFrameRate ) < 8) -> state_5 )
  {
  }
}

```

Fig. 14. CDL file generated from models

The presented approach has been tested and demonstrated on a UAV Video Streaming application described as a case study in this paper. The case study described a simple scenario; however, we have been able to model a much larger scenario in the developed modeling environment, with a higher degree of variability and adaptability. In our experience the simulation capabilities have been particularly helpful in fine-tuning the adaptation mechanism - one simple instance being the tun-

ing of the duration of staying in the duty and the test state using the simulation results in the presented case study.

The tool is still in the prototype state and several enhancements are planned. We plan to integrate a symbolic model-checking tool (e.g. Symbolic Model Verifier) for formal reasoning about the adaptation mechanism. With the aid of this tool we can establish various properties such as liveness, safety, reachability, etc., about the state-machine implementing the adaptation policy. We also plan to strengthen the computation and middleware modeling in order to facilitate analysis of the application and middleware components.

Acknowledgments

This work is supported by DARPA under the Program Composition of Embedded System (PCES) program within the Information Technology office (DARPA/ITO).

References

- [Bézivin, 01] Jean Bézivin, "From Object Composition to Model Transformation with the MDA," *Technology of Object-Oriented Languages and Systems (TOOLS)*, Santa Barbara, California, August 2001.
- [Harel, 87] David Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, June 1987, pp. 231-274.
- [Karr et al., 01] David Karr, Craig Rodrigues, Joseph Loyall, Richard Schantz, Yamuna Krishnamurthy, Irfan Pyarali, and Douglas Schmidt, "Application of the QuO Quality-of-Service Framework to a Distributed Video Application," *International Symposium on Distributed Objects and Applications*, Rome, Italy, September 2001.
- [Lédeczi et al., 01] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, November 2001, pp. 44-51.
- [Loyall et al., 01] Joseph Loyall, Richard Schantz, John Zinky, Partha Pal, Richard Shapiro, Craig Rodrigues, Michael Atighetchi, David Karr, Jeanna Gossett, and Christopher Gill, "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications," *IEEE International Conference on Distributed Computing Systems (ICDCS-21)*, April 2001, Phoenix, Arizona.
- [Neema, 01] Sandeep Neema, "System-Level Synthesis of Adaptive Computing Systems," Ph.D. Dissertation, Vanderbilt University, May 2001.
- [Nordstrom et al., 99] Greg Nordstrom, Janos Sztipanovits, Gábor Karsai, and Ákos Lédeczi, "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments," *International Conference on Engineering of Computer-Based Systems (ECBS)*, Nashville, Tennessee, April 1999, pp. 68-74.
- [Schantz and Schmidt, 01] Richard E. Schantz and Douglas C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," *Encyclopedia of Software Engineering*, Editors John Marciniak and George Telecki, Wiley and Sons, New York, 2001.