

A Model-Based Approach to System Specification for Distributed Real-time and Embedded Systems

*Radu Cornea¹, Shivajit Mohapatra¹, Nikil Dutt¹, Rajesh Gupta², Ingolf Kreuger², Alex Nicolau¹,
Doug Schmidt³, Sandeep Shukla⁴, Nalini Venkatasubramanian¹*

¹Department of Computer Science, UC Irvine,

²Department of Computer Science and Engineering, UC San Diego,

³Department of Electrical Engineering and Computer Science, Vanderbilt University,

⁴Department of Electrical and Computer Engineering, Virginia Tech.

1 Introduction

Distributed, real-time, and embedded (DRE) systems take input from many remote sensors, and provide geographically-dispersed operators with the ability to interact with the collected information and to control remote actuators. These devices are useful in a range of DRE application domains such as avionics, biomedical devices and telemedicine, remote sensing, space exploration and command and control. An important design challenge for such complex DRE computing systems is to satisfy performance and reliability constraints while ensuring efficient exploration through a very large architectural design space, and a very large implementation space for microelectronic system implementations. Current strategies in meeting these challenges has led to emergence of a new class of modeling and implementation tools that enable composition of such systems for microelectronic implementations and limited capabilities for retargeting existing compilers for new processors. However, the application development process for DRE systems continues to be very manually driven. The design of DRE systems is often very platform specific from the very early stages of the design process. Combinations of functional and non-functional requirements considered at the early stages of design are very specific for the system that is being developed. This makes the resulting system design inflexible (against changing requirements) and non-reusable (in settings with different requirements). To overcome these challenges, the FORGE project uses following three mechanisms for dealing with the specific challenges of DRE systems:

- Conceptualization and specification of DRE system requirements including those related to its structure, behavior, and performance/QoS guarantees;
- Modeling the available design knowledge (including the elicited requirements and the target platform) using flexible software architecture that are specified via architectural description languages (ADLs);
- Targeting flexible and optimizing middleware solutions and operating systems as our implementation platform.

Overall our approach is to use a model-based approach to system specification that allows reasoning about functional and non-functional properties of the system from the properties of the constituent components and the composition mechanism used; and to use a middleware infrastructure that lends itself to platform specific optimization for performance and size. Specifically, we focus on *adaptive* and *reflective* middleware services to meet the application requirements and to dy-

namically smooth the imbalances between demands and changing environments [Venkatubramanian01, Mohapatra02]. While a full discuss on the nature of middleware is out of scope here, very briefly, adaptive middleware is software whose functional and QoS-related properties can be modified either:

- (a) *Statically*, for example, to reduce the memory footprint, exploit platform-specific capabilities, functional subsetting, and minimize hardware/software infrastructure dependencies; or
- (b) *Dynamically*, for example, to optimize system responses to changing environments or requirements, such as changing component interconnections, power-levels, CPU/network bandwidth, latency/jitter; and dependability needs.

Reflective middleware [Wang01] goes a step further in providing the means for examining the capabilities it offers while the system is running, thereby enabling automated adjustment for optimizing those capabilities. Thus, reflective middleware supports more advanced adaptive behavior, i.e., the necessary adaptations can be performed autonomously based on conditions within the system, in the system's environment, or in combat system doctrine defined by operators and administrators.

Figure 1 illustrates the fundamental levels of adaptation and reflection that must be supported by middleware services: (a) changes in the middleware, operating systems, and networks beneath the applications to continue to meet the required service levels despite changes in resource availability, such as changes in network bandwidth or power levels, and (b) changes at the application level to either react to currently available levels of service or request new ones under changed circumstances, such as changing the transfer rate or resolution of information over a congested network. In both instances, the middleware must determine if it needs to (or can) reallocate resources or change strategies to achieve the desired QoS. DRE applications must be built in such a way that they can change their QoS demands as the conditions under which they operate change. Mechanisms for reconfiguration need to be put into place to implement new levels of QoS as required, mindful of both the individual and the aggregate points of view, and the conflicts that they may represent.

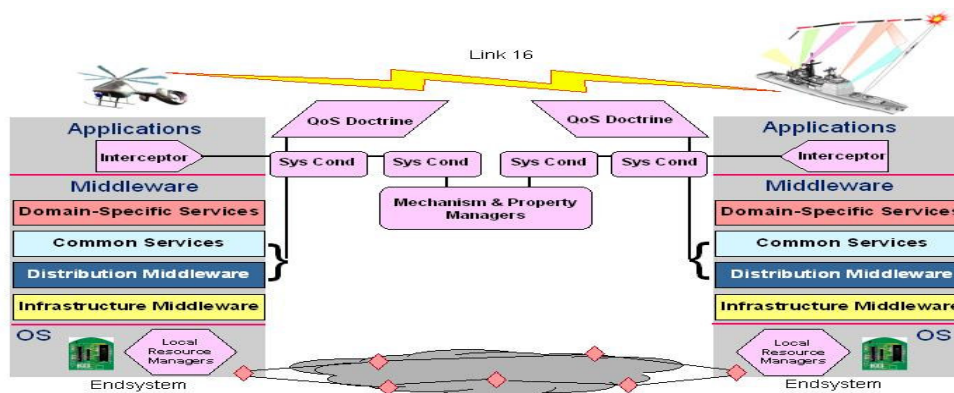


Figure 1: Middleware services for DRE applications

2 Collaborative Specifications and Semantic Objects

Recently advances in middleware technologies such as the ACE/TAO or real-time extensions to CORBA, have significantly improved the *decoupling* of implementation components within complex distributed, reactive systems. This decoupling helps building *component-oriented* applications that can be more flexibly composed, and are easier to configure. It facilitates integration

of off-the-shelf system parts, thus enabling better reuse of proven solutions. Furthermore, maintenance of component-oriented systems is simplified, because individual components can be more easily replaced than in tightly-coupled or monolithic system implementations.

At the heart of the decoupling of components in middleware infrastructures is a shift of focus from purely control-flow- and state-oriented system execution towards message- and event-oriented *component interaction* or *collaboration*. Event channels within the middleware, for instance, provide mechanisms for signaling the occurrence of events, as well as for the subscription to event notification. The behavior of the overall system emerges as the interplay of the components collaborating to implement a certain task or *service* by exchanging messages or processing events. Therefore, a crucial step in developing service-oriented systems is the capturing and modeling, as well as the efficient and correct implementation of the interactions among the components establishing and defining a service. The interactions in which a component is involved reflect the relationship between the component and its environment, and thus characterize the component's *interface* beyond static lists of method signatures.

MSCs and their relatives have been developed to complement the local view on system behavior provided by state-based automaton specifications. MSCs allow the developer to describe patterns of interaction among sets of components; these interaction patterns express how each individual component behavior integrates into the overall system to establish the desired functionality. Typically, one such pattern covers the interaction behavior for (part of) one particular service of the system. A particular strength of MSCs is capturing component collaboration transparently; hence these are ideal for representing the QoS constraints. Easiest examples are constraints related to timing and rate of events and actions, but other constraints – such as at most a certain amount of memory be spent in processing the enclosed message sequence – are also possible.

Streams and *Relations on streams* have emerged as an extremely powerful specification mechanism for distributed and interactive systems [Broy01], suitable for providing the semantic basis for MSCs and its hierarchical extensions. In this model, we view systems as consisting of a set of components, objects, or processes, and a set of named channels. Each channel is directed from its source to its destination component. Channels connect components that communicate with one another; they also connect components with the environment. Communication proceeds by message exchange over these channels.

Mathematically, a stream is a finite or infinite sequence of messages, occurring on a channel of the system. *In other words, streams represent the communication histories of system components.* Individual components can be understood as relations over their input/output histories. Intuitively, we describe a component by its reactions to the inputs it receives over time. This model lends itself nicely to the capturing of **collaborations** and services: they emerge as projections of the overall system behavior on certain components and their channels. Because in the stream model we can reference entire histories of component interactions, we can describe and reason about the global QoS properties mentioned above. Earlier, we have used this mathematical model successfully to define a precise service notion, supporting simple QoS specifications [Krueger02]. MSC specifications can be systematically transformed into state-oriented component implementations. This is an important step for ensuring that the implementation meets the elicited interaction requirements, and reduces the amount of manual work needed to achieve correct system designs.

Because collaboration specifications describe components *together* with the context they operate in, these provide important *design information* required to provide optimized system implementations. In particular, in the context of FORGE, a collaboration specification for a particular service will include information on what part of the middleware, and of other external components is required for implementing the service. An intelligent linker or runtime system can use this information to reduce the memory footprint of the implementation.

Architecture Description Languages (ADL) have traditionally been employed by the compilers for managing the micro-architectural resources of the CPU (registers, functional units). By specifying machine abstractions that capture both a processor's structure and behavior at a high level, the ADL can drive the automatic generation of compiler/simulator toolkits, allowing for early "compiler-in-the loop" design space exploration. This simultaneous exploration of the application, architecture and compiler allows early feedback to the designers on the behavior of the match between the application needs, the architectural features, and the compiler optimizations.

In case of heterogeneous machines, comprising multiple processors/resources, the ADL alone is not sufficient. A *Resource Description Languages* (RDL) is included for specifying attributes of the available hardware, as well as communication structure, constraints and system requirements. The RDL/ADL mechanisms expose both the architecture of the system and the resource constraints to the compiler, allowing for an effective match of the application to the underlying hardware.

An example of ADL language that supports fast retargetability and DSE is EXPRESSION [Express, Halambi99, Mishra01]. The language consists of two main parts: behavior and structure specification. In the behavior component the available operations and instructions are enumerated, as well as their execution semantics; this drives both the code generation phase of the compiler and the functional execution part of the simulator. The structure component describes the components of the processor, the connectivity (paths) between them and the memory subsystem; it is mainly used in the compiler's optimization phases to generate code that best match a given architecture. Also, the simulator accurately generates cycle by cycle statistics based on the same description.

4 Resource and Architecture Description in FORGE

Compiler technology has traditionally targeted mainly individual processors or controllers. However, designers of DRE applications increasingly must deal with large systems, containing multiple processor cores, peripherals, memories, connected through different wireline and wireless networks. FORGE extends the traditional notion of a compiler and ADL to include not only platform specific architectural details but also capabilities of the middleware services needed for an application. By viewing the system components, such as CPUs (e.g., data collection drones), different computing devices and peripherals as resources that are described in a high-level language, it is possible to allow the compiler to generate service specifications in a more globally optimal manner.

The compiler is not the only place where the higher-level information can prove useful. At runtime, information can be passed between abstractions at different levels, making the decision process more aware of the actual device and application, so that the power and resource utilization are improved. The OS/hardware level resides closest to the actual hardware device and has full knowledge of its capabilities and limits. By relaying some of this information to the higher levels, it helps the middleware framework in making decisions of task migration and node/network restructuring (better mapping between tasks with different demands and available heterogeneous nodes):

- Computing power (expressed in MIPS): in general, nodes are heterogeneous; their hardware processing capabilities may vary over a large range depending on the type and number of processors, frequency at which they are running and other local factors. Similarly, tasks (particularly in time-constrained applications) may be characterized for WCET as well as response time requirements. Middleware can make use of this information when migrating the tasks between nodes or duplicating tasks on a node in response to increases in the processing workload.

- Available total memory: memory availability may not only be different but also diverse across different nodes in a DRE system. Memory availability can be matched to task memory footprints for improved resource utilization.
- Availability of specialized functional units (and resources)
- Power budget or efficient battery discharge profile: typically, the middleware level assume a fixed energy available to each node and a linear discharge rate. In reality, the available energy depends heavily on the discharge curve. Discharges under a high current may weaken the battery and shorten its life. Similarly, other nodes may be able to renew their energy levels by using solar cells. In this case, the available energy profile on the node will have a periodic behavior, with high level during daytime and lower levels during nights, when the only energy comes from the battery. A coordinated attempt to match desired energy consumption profile through distribution and scheduling of tasks in a DRE system would be desirable for efficient utilization of system resources.

Thus, the lower level (OS/hardware) can provide the middleware levels with a simplified view of the current state of the actual device at any point in time (in terms of processing capabilities, available power and memory). On the other end, the application/middleware levels controls how the application is to be distributed and run on the available network of nodes. They have an extended view of the requirements of the system at any point of time and can provide valuable information to the OS/hardware (lower levels). When making scheduling decisions, the OS/hardware makes no assumption about the current and future position of the node or upcoming processing requirements. They have a limited local view, where the actual distributed system is not visible. When useful, the middleware level can make parts of the global information which are relevant available to the lower levels:

- at the OS level, a task is viewed as a black box, assuming the worst case execution time and maximum power consumption. In reality, a task may be profiled a priori and a more realistic execution time and power profile could help the OS to make better scheduling decisions.
- some tasks are not too important for the system at specific times and they may be run at the lowest rate (even serialized). The OS level will make decisions to slow down the processing greatly reducing the power consumption (either by DVS or DPM).
- if the middleware level has a better understanding of the application flow (for instance, the processing starts slowly, and it is not very critical, but later the computation can increase heavily, pending the activation of an observed sensor), it can instruct the lower levels to save energy, i.e., if possible run the tasks at lower QoS (such as a lower frame rates for a frame based stream processing), or serialize the computation.

Figure 2 shows our vision of the application development model for complex heterogeneous distributed computing platform. The Lowest layer shows an abstract picture of a heterogeneous computing platform consisting of multiple devices connected over wire-line/wire-less communication links. The second layer shows the Architecture description of such a platform, and resource constraint description of the application requirements (such as power budget, real-time constraints etc.). The compiler (to be developed), takes the application functional specification, the ADL, and RDL, generates the services, and middleware configuration shown in the second highest layer, and their deployment information across the platform.

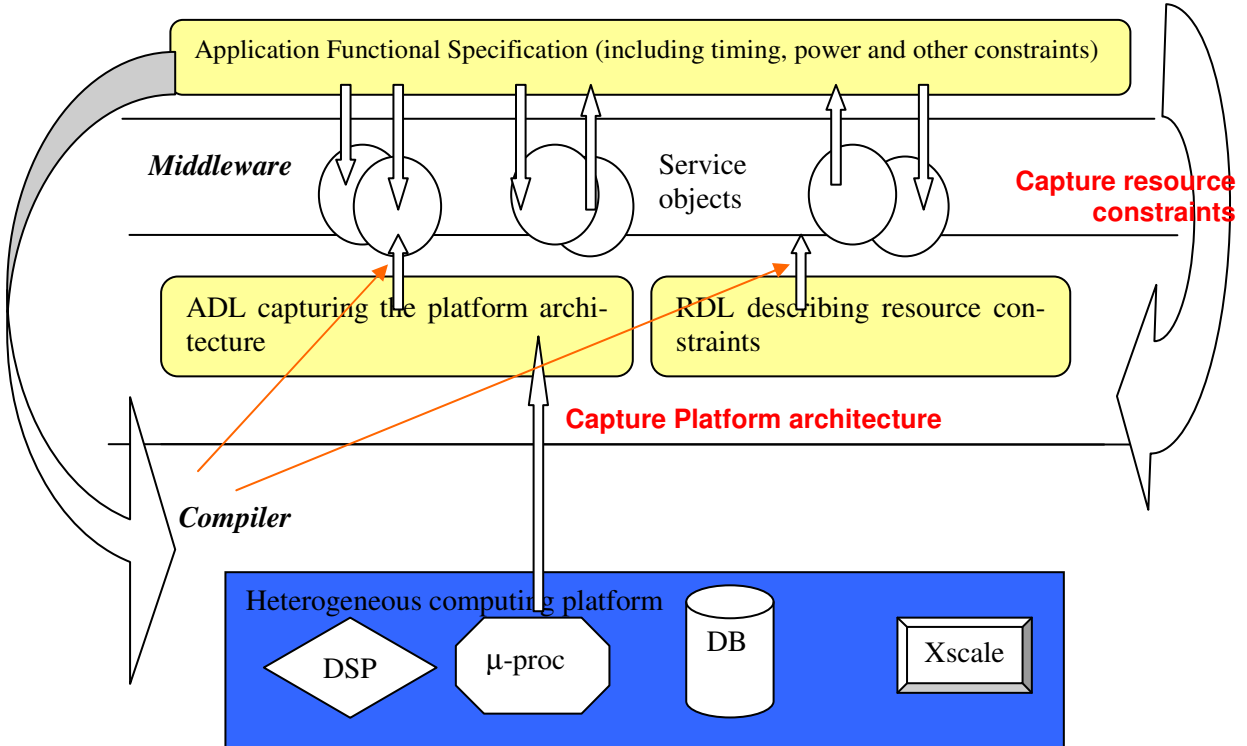


Figure 2: Application Development Model for Distributed Real-time Embedded Systems

Compiler-Runtime Interaction and the Role of RDL

Our approach towards integration of the compiler with the runtime system relies on user-level creation and management of threads (parallelism), and a new application-OS kernel interface that supports a simple communication protocol between the OS and the executing application. This communication protocol between user and kernel does not require context switching and hence it involves minimal overhead. In fact, the critical aspects of the user-kernel interface and communication protocol consist of posting and inspecting processor and thread counts from both the client (application) and the server (OS), and resuming execution (suspended threads) from user-space without the involvement (and hence the cost) of the OS kernel. Both of these interfaces can be implemented by inexpensive memory operations (loads/stores) to a shared region of the virtual address space, which is pinned in physical memory. In the proposed approach, the OS decides (according to policies that we have already developed), when and how many processors to give to a process and for what duration. This information is communicated through the shared region of memory (via an API). From the time of allocation, each processor is managed explicitly by user-code, de-queuing and executing user-level threads while possibly queuing other threads.

Our approach eliminates the traditional manual partitioning of functionality of the application into services/objects, and automates the generation of such partition and mapping. By definition, the compiler has the most detailed information about applications, and coupled with target computing platform description, it becomes the tool of choice for generating and inlining the runtime support code into the application code. This can be done by partitioning the program into independent threads exploiting user-specified or compiler extracted loop and task parallelism, and (using the RDL target description) translating data and control dependencies into sequencing and communi-

cation code which, during execution, renders runtime support to the user application. In addition, the compiler can also generate all the necessary data structures including scheduling queues and designated space for user-level context save/restore operations. Since generation of the runtime system is automated and inlined in user code, there is opportunity for compile-time optimization of the runtime system code itself, which is impossible in traditional approaches.

5 Case Studies

We outline here two applications from different domains to demonstrate our approach to DRE software development.

- The first example is based on automated target recognition (ATR) systems. The ATR application consists of two main components: target detection and target recognition. One level down, the application processing part can be divided into four main tasks, which operate independently on groups of frames: TARG (target detection), FFT (filters), IFFT (filters), DIST (compute distance) shown in **Figure 3**. This division allows OS scheduler to parallelize the tasks into a pipelined version amenable to deployment in a distributed environment.

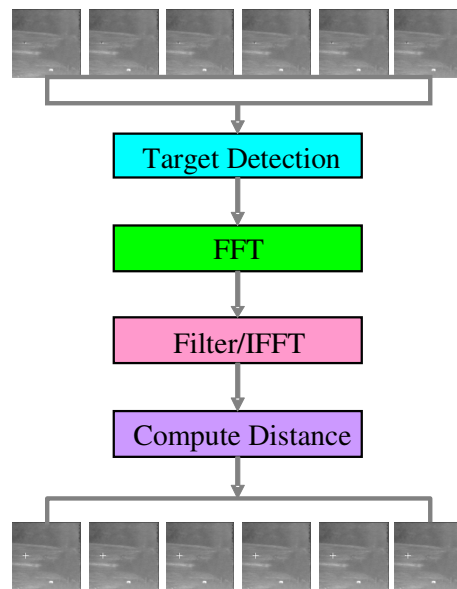


Figure 3: ATR Application Structure

We can envision an environment with hundreds of nodes performing target recognition, geographically spread over a large area and running on a heterogeneous network of hardware devices. The devices may be small drones, capable of moving on flat terrain, small unmanned planes, or even miniature sensors that are dropped from airplanes over large areas. The nodes communicate wirelessly; some may have limited range of communication and may require a proxy node in order to relay the information to control nodes.

Operating such a complex distributed application is not trivial and it requires cooperation between many components and different layers of the abstraction view (middleware,

hardware). Because of the extremely heterogeneous operating environment, there are many decisions to be made in order to preserve the functionality of the application and to meet the QoS requirements at the same time. Through a tight middleware / hardware level integration, the complexity of such a system is reduced to manageable proportions.

- The second example studies the interaction between middleware and hardware levels in the context of quality driven, power-aware video streaming to mobile devices. The application runs in a distributed environment consisting of application servers, proxy servers, access points and clients (mobile wireless devices, PDAs) as depicted in **Figure 4**. The proxy nodes are intermediary processing servers that control the way content providers are streaming video multimedia to clients. Their job is to relay the video stream to clients, after optimizing it for the particular QoS the client requests, through a transcoding process.

In response to actual status of a particular device (periodically reported by the hardware level on the client), the middleware layer, which controls the transcoding at the proxy decides on the optimal stream quality to be generated. The final objective is that of maximizing the end user experience at the client device. In this particular case, best experience would mean the possibility of watching a movie on the device for the entire duration, at the best quality as permitted by the residual power available in the battery (checked periodically). The objective here is not minimizing the power consumption, but providing the user with the best possible user experience. By integrating middleware techniques with hardware level optimizations, such a tradeoff between quality and power can be easily attained.

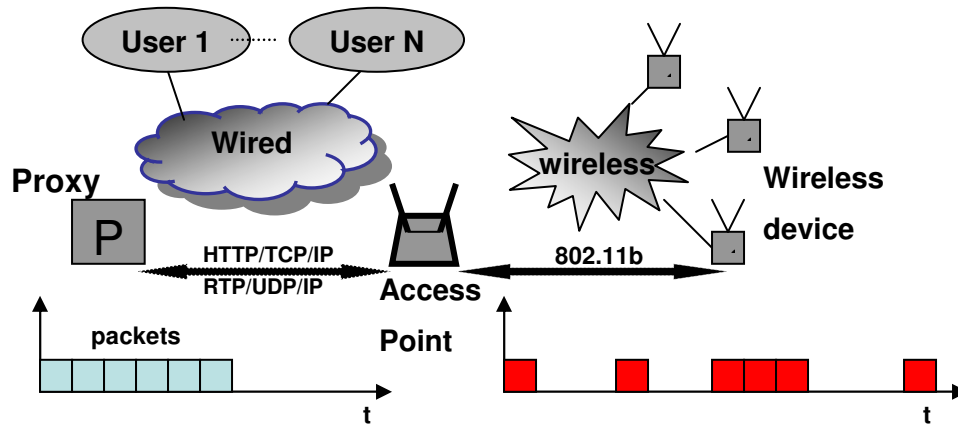


Figure 4: Proxy-based video streaming

6 Summary

FORGE brings together a number of advances in architectural modeling, software architecture and distributed/real-time systems to build a platform that provides two fundamental capabilities for DRE system development: (a) conceptualization and coding of the design knowledge through collaborative specifications that are inherently matched to distributed solutions; and (b) exploitation of the design knowledge across all development phases for the DRE systems. Our proof-of-concept FORGE prototype is built upon collaborative specifications captured by extensions to the message sequence charts (MSCs) that drive the customization of CompOSE middleware services and generate node-architecture specific code through descriptions of the architecture and resources captured using ADL and RDL respectively.

References

[Azevedo02] Ana Azevedo, Ilya Issenin, Radu Cornea Rajesh Gupta, Nikil Dutt, Alex Veidenbaum, Alex Nicolau, "*Profile-based Dynamic Voltage Scheduling using Program Checkpoints in the COPPER Framework*", Design Automation and Test in Europe, March 2002.

[Broy01] M. Broy, K. Stølen, "*Specification and Development of Interactive Systems*". Focus on Streams, Interfaces, and Refinement. Springer, 2001

[Chou02] P. H. Chou, J. Liu, D. Li, and N. Bagherzadeh, "*IMPACCT: Methodology and tools for power aware embedded systems*", Design Automation for Embedded Systems, Kluwer International Journal, Special Issue on Design Methodologies and Tools for Real-Time Embedded Systems, 2002.

[Express] Expression Project Webpage, <http://www.cecs.uci.edu/~aces/projMain.html#expression>

[Halambi99] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, Alex Nicolau, "*EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Re-targetability*", DATE 99.

[Havinga00] P. J. M. Havinga, "*Mobile Multimedia Systems*", PhD thesis, University of Twente, Feb 2000.

ITU-R Recommendation BT-500.7, "*Methodology for the subjective assessment of the quality of television pictures*", ITU Geneva Switzerland, 1995.

[Krueger02] I. Krüger, "*Towards Precise Service Specification with UML and UML-RT*", in: Critical Systems Development with UML (CSDUML). Workshop at «UML» 2002, 2002

[Leeser97] P. Soderquist, M. Leeser, "*Optimizing the data cache performance of a software MPEG-2 video decoder*", ACM Multimedia, 1997, pp. 291–301.

[Mishra01] Prabhat Mishra, Peter Grun, Nikil Dutt, Alex Nicolau, "*Processor-Memory Co-Exploration driven by a Memory-Aware Architecture Description Language*", the 14th International Conference on VLSI Design (VLSI Design 2001), Jan. 2001

[Mishra02] Prabhat Mishra, Hiroyuki Tomiyama, Ashok Halambi, Peter Grun, Nikil Dutt, Alex Nicolau, "*Automatic Modeling and Validation of Pipeline Specifications driven by an Architecture Description Language*", VLSI Design ASPDAC 2002.

[Mohapatra02] Shivajit Mohapatra, Nalini Venkatasubramanian, "*A Distributed Adaptive Scheduler for QoS Support in Compose|Q*", Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2002). January 7-9, 2002 San Diego, CA

[Mohapatra03] Shivajit Mohapatra, Nalini Venkatasubramanian, "*PARM: Power Aware Reconfigurable Middleware*", IEEE International Conference on Distributed Computer Systems (ICDCS-23), Rhode Island, May 2003.

[Radkov03] P. Shenoy, P. Radkov, "*Proxy-Assisted power-friendly streaming to mobile devices*", MMCN, January 2003.

[Schantz02] Richard E. Schantz and Douglas C. Schmidt, "*Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications*," Encyclopedia of Software Engineering, Wiley and Sons, 2002.

[Venkatubramanian01] Nalini Venkatubramanian, Mayur Deshpande, Shivajit Mohapatra, Sebastian Gutierrez-Nolasco and Jehan Wickramasuriya, "*Design & Implementation of a Composable Reflective Middleware Framework*", IEEE International Conference on Distributed Computer Systems (ICDCS-21), Phoenix AZ, April 2001.

[Wang01] Nanbor Wang, Douglas C. Schmidt, Kirthika Parameswaran, and Michael Kircher, "*Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications*," IEEE Distributed Systems Online special issue on Reflective Middleware, 2001.