FORGE: A Framework for Optimization of Distributed Embedded Systems Software[*]

Radu Cornea[1], Nikil Dutt[1], Rajesh Gupta[2], Ingolf Krueger[2], Alex Nicolau[1],Doug Schmidt[3], Sandeep Shukla[4]

[1]Department of Computer Science, UC Irvine,
[2]Department of Computer Science and Engineering, UC San Diego,
[3]Department of Electrical Engineering and Computer Science, UC Irvine,
[4]Department of Electrical and Computer Engineering, Virginia Tech.

1 Introduction

New and planned commercial and military **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. With the continuing advances in the computational power, and reductions in the costs of high-performance processing and memory elements, we are beginning to see proliferation of portable devices (e.g., intelligent sensors, actuators, and sensory prosthetics) with substantial processing capabilities. 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. Examples of domains that could benefit from microelectronic SOC advances include: automated transportation systems, distance learning, telemedicine, analysis for combat situations, video conferencing, virtual reality simulation, and weather forecasting and analyses.

In circumstances where the presence of a human in the loop is either too expensive or too slow, these systems must also respond autonomously and flexibly to unanticipated combinations of events at run-time. Further, these systems are increasingly networked to form long-lived "systems of systems" that must run unobtrusively and autonomously, shielding operators from unnecessary details, while simultaneously communicating and responding to mission-critical information at heretofore infeasible rates. As DRE applications grow more complex, so are the embedded computing devices that have evolved at the rate of Moore's law from the Von Neumann era of sequential computing, to the era of supercomputers, to workstation clusters, to multi-processor servers, all the way to multi-threaded multi-processor system on chip. Developing applications for such complex DRE systems has grown more difficult as the software productivity gap, i.e., the gap between software development productivity and complex hardware availability, has increased. Further, increasing communication and network capabilities are driving an unprecedented surge in heterogeneous platforms with multiple computers (some times in enormous numbers) connected via network (wireless or wireline), all connected in a web of embedded devices, handheld, and otherwise.

> Consider, for instance, network management [Udupa99] for a multi-layered data/telecommunication network. Such a system consists of diverse network elements, such as telecommunication switches, routers, network devices, control-center servers and clients, field workers' handheld computers etc. that must be managed together as a unified network. End-to-end network fault management and performance management is crucial for these networks' operation and QoS (Quality of Service) guarantees. Currently such network management systems are programmed at a low level of abstraction. This low abstraction in viewing such a complex

[*] This work was partially supported by NSF award ACI-0204028.

DRE system requires the application developers to think of each device and the services hosted on them as separate service/software elements that communicate via middleware. A middleware is a set of software layers that functionally bridge the gap between (1) application programs and (2) lower-level underlying operating systems and network protocol stacks to provide services whose qualities are critical to DRE systems [Schantz02].

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 reason why existing DRE development process is manually driven is not hard to find: the software architecture for complex computing architectures with distributed and diverse computational capabilities is not yet developed. FORGE is an application development for DRE systems that will enable an application developer to structure and design software for platforms that may not exist or may be concurrently developed. The key to our approach is a systematic method to model architectures and a fundamental rethinking of the division of static versus runtime delegation of functionality. In this paper, we describe the architecture of FORGE and specific innovations being pursued. This paper is organized as followed: after a brief overview of the FORGE architecture in Section 2, we describe DRE system specification and modeling of both functional as well as non-functional aspects (properties and constraints) in Section 3, followed by a discussion of the middleware capabilities and their optimizations for specific DRE platforms in Section 4. We then consider the link to compiler techniques and component APIs using interface description languages (IDLs) in Section 5. We conclude by presenting a case study of an application.

2 DRE System Design using FORGE

The design of complex systems has often been extremely 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, we propose the following three mechanisms for dealing with the specific challenges of DRE systems as outlined in the introduction:

- 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 dynamically smooth the imbalances between demands and changing environments. 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.

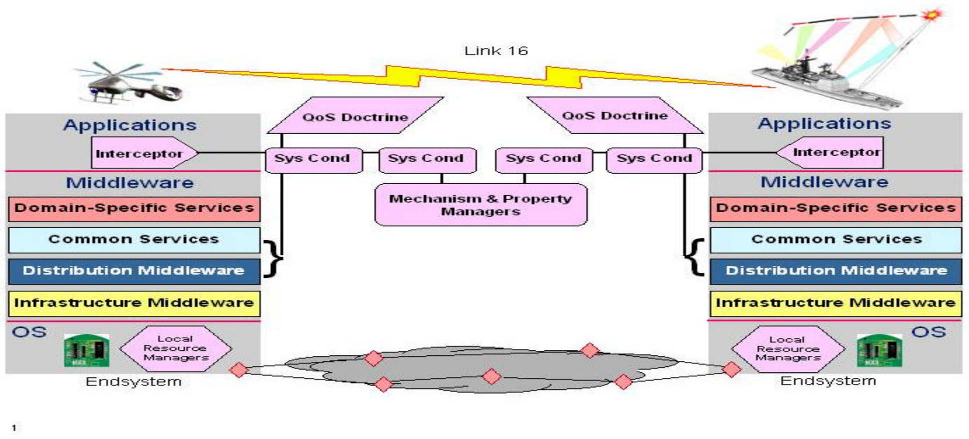


Figure 1: ***Middleware services for DRE applications***

A Model-based Approach to System Specification

Using a model for describing architectures – a model that treats different design-concerns independently, but also combines these concerns into a coherent model – can also enhance reusability. In our view, higher level conceptualization is more effectively carried out by reasoning through the collaborative aspects (rather than compositional aspects, which is an implementation issue) of the design. The methods to capture interactions between components have been explored earlier – most prominently in the context of asynchronous component interaction by means of sequence diagrams – these are not enough to capture dynamic control behavior and quality constraints. Message Sequence Charts or MSCs can be used to not only capture component collaborations but also timing constraints on such interactions [Krueger00]. We build upon these to capture both functional and non-functional aspects. In the sequel, we use the term ***Semantic Object*** to indicate a meta-model of components, interfaces and services that are needed in DRE systems. The semantic objects are generated from a formal model, called ***streams***, that is described later in Section 3 below. The stream model can capture the collaborative aspect of services that a designer may specify using suitable notational diagrams, charts or mathematical relations.

3 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. As an example, consider the specification of a fictitious video streaming service as described in **Figure 2**. Here, we have shown the collaboration between the Video Source, the Video Distributor, and Video Display Hosts to transmit sequences of video frames in a simplified notation related to MSCs.

This diagram conveys information on both the distribution structure, and the coordination of the components. The sequence diagram at the top shows how the three components interact to establish the service. The augmentation shows timing constraints which define quality constraints on the participating components. The Video Distributor, for instance, needs to provide adequate mechanisms for memory, thread pooling, message prioritization, and power consumption to meet the QoS constraints specified in the diagram.

Our approach is to find automatic ways to identify and optimize needed middleware services. For instance, the sequence diagram defines precisely which messages the components exchange, and in which order they must occur; these are important functional aspects of the component interfaces.

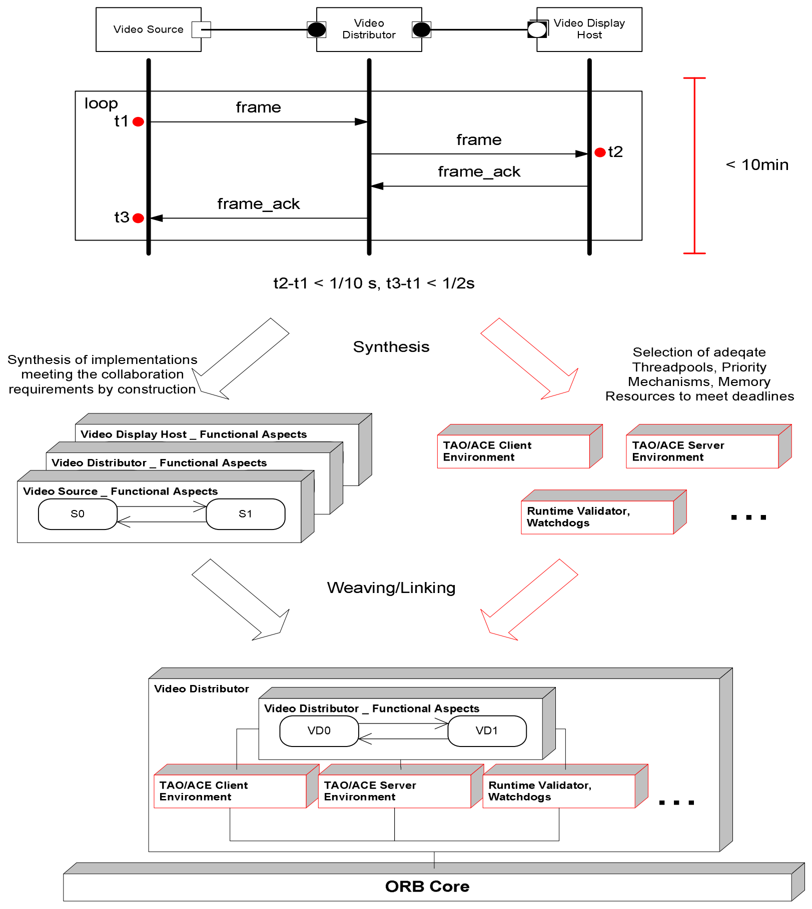

Video Source
Video Distributor
Video Display Host
loop
t1
frame
frame
t2
frame_ack
frame_ack
t3
< 10min
t2-t1 < 1/10 s, t3-t1 < 1/2s
Synthesis
Synthesis of implementations meeting the collaboration requirements by construction
Selection of adeqate Threadpools, Priority Mechanisms, Memory Resources to meet deadlines
Video Display Host _ Functional Aspects
Video Distributor _ Functional Aspects
Video Source _ Functional Aspects
S0
S1
TAO/ACE Client Environment
TAO/ACE Server Environment
Runtime Validator, Watchdogs
. . .
Weaving/Linking
Video Distributor
Video Distributor _ Functional Aspects
VD0
VD1
TAO/ACE Client Environment
TAO/ACE Server Environment
Runtime Validator, Watchdogs
. . .
ORB Core

Figure 2: ***MSCs can be used to capture functionality, coordination and timing aspects, (design adapted from Karr, et al, in ACM Multimedia 2001.)***

To meet the nonfunctional aspects – such as the global bound on transmission time of 10 minutes stated in the diagram – configuration of the proper execution environments (regarding performance, channel throughput, scheduling etc.) depends on the capabilities of the underlying middleware. By analyzing the performance requirements for an entire collaboration we could, for instance, identify, synthesize, or configure automatically the adequate client and server environments in ACE/TAO, as well as corresponding runtime-validators, watchdogs, and similar mechanisms. The functional and non-functional aspects thus collected can be weaved together to yield a RT CORBA implementation meeting both classes of requirements.

Streams and ***Relations on streams*** have emerged as an extremely powerful specification mechanism for distributed and interactive systems [Broy01, Broy98], 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 [Krueger02a, Krueger02b]. 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 an ADL language that supports fast retargetability and DSE is EXPRESSION [Express]. 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 run-time, 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 3 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.

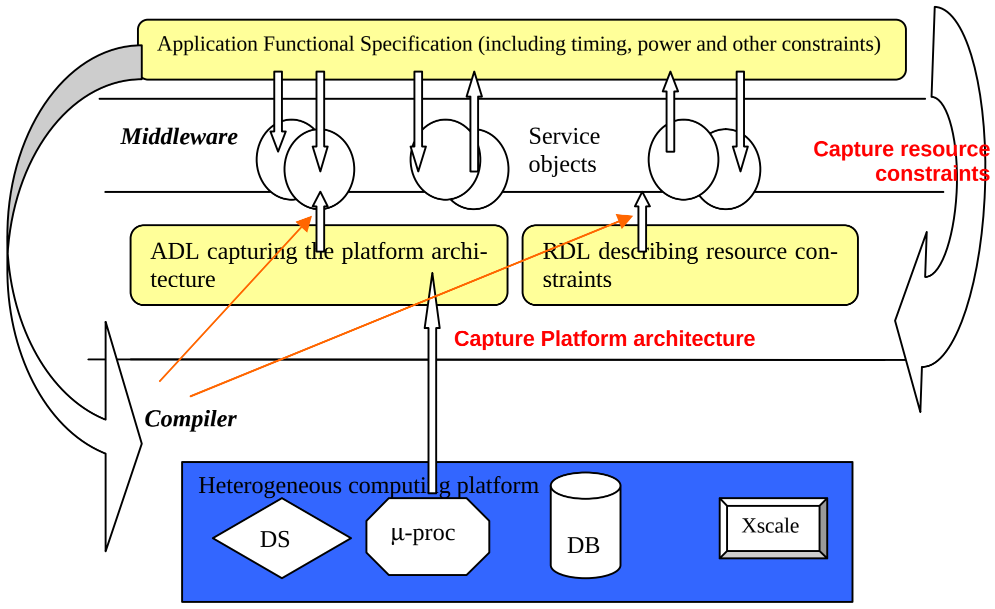

Application Functional Specification (including timing, power and other constraints)
Middleware
Service objects
Capture resource constraints
ADL capturing the platform architecture
RDL describing resource constraints
Capture Platform architecture
Compiler
Heterogeneous computing platform
DS
μ-proc
DB
Xscale

Figure 3: ***Application Development Model for Distributed Real-time Embedded Systems***

Compiler-Runtime Interaction and the Role of RDL

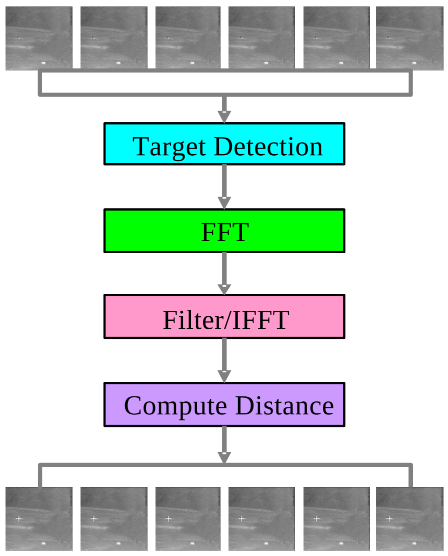
Our approach towards integration of the compiler with the runtime system relies on user-level creation and management of threads (parallelism), and a new application-OS kernel interface that supports a simple communication protocol between the OS and the executing application. This communication protocol 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 communication 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.

Another important advantage of this approach is the ability to match the amount of parallelism (number of threads) generated by the application at runtime with the number of computing resources available to the application at each step during execution. We can achieve his by relying on the Hierarchical Task Graph (HTG) [Girkar92], an intermediate parallel program representation which encapsulates minimal data and control dependences, and which can be used for the extraction and exploitation of functional or task-level parallelism. The hierarchical nature of the HTG facilitates efficient task-granularity control, and thus applicability to a variety of parallel architectures. The HTG captures the explicit hierarchy of the underlying computation so that nested loops, for example, are represented by compound nodes at different levels of the representation. The actual number of threads allocated to each loop nest is determined at runtime as control reaches that loop. A simple inspection of the allocated number of processors through the OS API can therefore guide the runtime system in making the most appropriate decision w.r.t. the exploitable degree of parallelism at each level of the computation. In this manner, the compiler produces *scalable code* – application code that is compiled once but can execute on different node architectures and generate different numbers of parallel threads matching the available resources in each case. Besides the obvious benefits of such a capability on the performance of individual applications, such scalable codes allow a seamless integration of multiprogramming, time-sharing and real-time processing on one hand, and parallel processing on the other; or equivalently, a cooperative fusion of time- and space-sharing on shared and distributed memory systems.

5 A Case Study

While the project is in very early stages, we outline here an application from automatic target recognition (ATR) code to demonstrate our approach to DRE software. The ATR application consists of two main components: target detection and target recognition. Going down one level, the main 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 4.**



(a)

```
(Application ATR
  (Contains TARG FFT IFFT
DIST)
  (Paths
    (TARG FFT)
    (FFT IFFT)
    (IFFT DIST)
  )
  (Deadline 16ms)
  ...
  (Task TARG
    (FloatingPoint NO)
    (Scalable YES)
    (Memory 1Mb)
    ...
  )
  (Task FFT
    (FloatingPoint YES)
    (Scalable YES)
    (Memory 1Mb)
    ...
  )
```

(b)

Figure 4: ***Specification of ATR Application Structure***

This division allows OS scheduler to parallelize the tasks into a pipelined version; from here, different decisions can be made based on the constraints imposed by the application (e.g.: processing one frame through the pipeline should not take more than 16 ms). Each task has its own power requirements and completion time. By running the hardware under different operating modes or by the use of dynamic voltage scaling, the task scheduler in FORGE can minimize the power consumption while still meeting timing constraints [Chou02]. Another scenario is scheduling the tasks so that the power consumption follows an available power profile (and meets the timing constraints) [Azevedo02].

By taking this simple application one level up, into a distributed world, we can imagine an environment in which there are hundred 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 a large hard reachable area. Most of these nodes communicate wirelessly; the small ones have limited range of communication and require a proxy node in order to relay the information to control nodes. Proxies could be dedicated, or, when needed, any node with sufficient processing power can act as a proxy for neighbors within communication range that are not capable of it.

The target recognition is based on images coming from light sensors. Part of the nodes has sensors operating on visible spectra, while others operate in infrared and are able to continue the tracking even during night. The terrain is predominantly flat, with some dispersed hills where only airborne sensors can follow. The application's main task it to track the targets over the observed area. Operating such a complex distributed application is not trivial and it requires cooperation between many components and different layers in our simplified abstraction view. 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.

At the middleware/application layers, we seek to enable adaptation of components and services to demands and changing environments using reflective middleware services []. Very briefly, reflection is the capability to automatically reconfigure structure of the distributed computation by migrating components from one node to others, reshaping the topology of the network or replicating/dereplicating nodes. In this context, we are using CompOSE|Q reflective distributed middleware infrastructure [Nalini01]. Based on a two level meta-architecture, it supports all the required base services for replication, dereplication, migration and other higher level services. The core runtime resides on every node; it implements meta-level services, manages system resources and controls the runtime behavior of application (base) level services. If necessary, part of the application level components can be migrated (offloaded) between nodes, to free up constrained nodes and allow for more efficient operation. This requires efficient exchange of key pieces of information from lower levels. **Figure 5** shows the information captured within FORGE specification.

```
(Node MOBILE1
  (Processor 400MIPS)
  (Memory 32Mb)
  (DPMCapable NO)
  (DVSCapable YES)
  (DVSModes
    (m0 600Mhz 2.2V)
    (m1 500Mhz 1.8V)
    (m2 400Mhz 1.5V)
    (m3 300Mhz 1.1V)
  )
  (PowerSource
    (Battery 50Wh)
    (SolarCell 5Wh
      (Period 24h)
      (Duration 9h)
    )
  )
)
```

```
(TaskProfile
    (Task TARG
      (m0 0.66ms 7W)
      (m1 0.79ms 4W)
      (m2 0.99ms 2W)
      (m3 1.32ms 0.9W)
    )
    (Task FFT
      (m0 0.29ms 6W)
      (m1 0.34ms 3.5W)
      (m2 0.43ms 1.8W)
      (m4 0.57ms 0.75W)
    )
    ...
  )
  (Sensors
    (Video
      (Spectra Visible)
    )
  )
  ...
)
```

```
(Node MAIN1
  (Processor 800MIPS
800MIPS)
  (Memory 1000Mb)
  (DPMCapable NO)
  (DVSCapable NO)
  (PowerSource
    (Line NOLIMIT)
  )
  (TaskProfile
    ...
  )
```

Figure 5: ***Node information captured in FORGE***

There are two parts to the specification: (a) the application part shown in **Figure 4(b)** describes the task decomposition and system level constraints. For instance, the ATR application is composed of four main tasks (TARG, FFT, IFFT, DIST), connected in series (the 'Paths' section). In this simplified case, there is only a global deadline: for processing one data input element (frame). Each task is characterized next, including special units (floating point) and memory requirements, scalability (capability of running multiple instances in parallel, to increase workload) and other information relevant at this level. (b) The node description abstracts node-level capabilities including (as applicable) task profiles for different operating modes for both time and power consumption. For instance, in **Figure 5**, the first node, MOBILE1, has a dynamic voltage scaling capable processor, that can operate in four modes (m0 to m3), each at different frequency and voltage. The node has a processing power of 400MIPS, 32Mb available total memory and two power sources: battery and solar cell. For each task timing and power profiles when running on the current node in different power modes are included. The second node is a mainframe with two processors and 1Gb of main memory. It has unlimited power budget (i.e. power line connected) and is not capable of DPM or DVS. While greatly abstracted, this information can still allow a range of middleware optimizations such as:

- the ATR application consists of two main parts: target detection and target recognition. Target detection is integer based, while target recognition makes use of floating point while applying FFT and IFFT filters. When deciding to offload parts of the computation to a proxy node, the middleware layer may decide between two or more available proxy nodes depending on their floating-point computation capabilities.
- while in observing mode, most of the nodes (drones) operate in a saving power state (low QoS). Once a target is being identified by any of the active drones, the nodes in the surrounding area are immediately notified by the control layer and the local OS decides on switching to the normal operating mode, at full voltage and frequency. If a node has multiple sensors or higher processing capabilities, the middleware layer may decide on using them as proxy for the small nodes by offloading some computation or middleware components.
- nodes may be able to perform under different power modes; others may allow dynamic voltage scaling. When the night comes, the controlling layer may decide to migrate some of the processing to DVS capable drones and shut down the ones operating on visible light in order to save power.
- if the tracked target is moving onto rough or high terrain, where the mobile drones cannot follow, the nodes operating from planes are notified and the whole network is restructured to allow uninterrupted tracking. If the area is already covered with small sensor nodes, which were dropped from planes, are not capable of motion and have limited communication range, the blocked mobile nodes which are close by may act as proxy nodes for the middleware communication.
- many drones have a limited power budget which is made available by the hardware level to the middleware control level. Based on this, the middleware knows at any point what is the distribution of the available power with respect to the nodes in the network and may instruct a reconfiguration of the network so that a good coverage of the area is obtained and the nodes will be able to perform the tracking even if some of them will run out of energy.
- in time, some drones may experience malfunctions of internal subsystems (e.g. the light sensor may stop functioning, or the motion engine may fail due to bad road conditions). The hardware level will then inform the middleware of the situation, and a decision can be made: the sensor less node may act in the future as a processing only node (or proxy), by situating itself among active drones and the stuck drone can act as a fix sensor.
- depending on the tradeoff between node capabilities and uplink power demand, the higher control level may require from a drone to upload the data directly from the sensor, without preprocessing. When many drones are closed by over a small area, because inter-node communication can be done with less power, most of the computation can be done locally by distributing the work and only one upstream of data will be necessary, potentially saving power.

This example shows us that a global view of the system at all levels of abstractions, with information flowing between different levels is beneficial for setting up and managing an application a distributed environment. The high-level system specification can drive the compiler to automatically generate not only the different components of the applications, but also the required middleware level services and interface that together glue the system components and help towards performing the overall functionality. Dynamically, the run-time system can base its decisions on the high-level specification and make better tradeoffs when confronted with changes in the environment that requires a redistribution of computation or network reconfiguration.

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|Q middleware services and generate node-architecture specific code through descriptions of the architecture and resources captured using ADL and RDL respectively.

References

[Udupa99] Divakara K. Udupa *"TMN: Telecommunications Management Network"*, McGraw-Hill Professional Publishing, January 1999.

[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.

[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.

[Krueger00] I. H. Krüger, *"Distributed System Design with Message Sequence Charts"*, Dissertation, Technical University of Munich, 2000, online at: http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2000/krueger.html

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

[Broy98] M. Broy, I. Krüger, *"Interaction Interfaces - Towards a scientific foundation of a methodological usage of Message Sequence Charts"*, in: J. Staples, M. G. Hinchey, Shaoying Liu (eds.): Formal

[Krueger02a] I. Krüger, *"Specifying Services with UML and UML-RT"*, Electronic Notes in Theoretical Computer Science, Vol. 65 (7), 2002

[Krueger02b] I. Krüger, *"Towards Precise Service Specification with UML and UML-RT"*, in: Critical Systems Development with UML (CSDUML). Workshop at «UML» 2002, 2002

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

[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.

[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.

[Nalini01] Nalini Venkatasubramanian, 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), April 2001.

[Girkar92] Milind Girkar, Constantine D. Polychronopoulos, *"Automatic Extraction of Functional Parallelism from Ordinary Programs"*, IEEE Transactions on Parallel and Distributed Systems 3(2): 166-178, 1992.