A Real-Time Tool Set for the ARTS Kernel

Hideyuki Tokuda and Makoto Kotera

Abstract

Advances in Software Engineering provided us with a set of modern programming tools for building a large complex software system. However, we cannot simply reuse the existing tool set for designing and building complex real-time computing systems without considering the time management capabilities. In particular, in real-time systems, nasty "timing bugs" are very difficult to capture and eliminate from the system. Our approach is to build an integrated tool set which can reduce the complexity of the development of real-time software. The tool set should be able to detect and remove the potential "timing bugs" at the system design phase rather than at the system testing and debugging phase.

In this paper, we describe an integrated real-time tool set. It consists of a schedulability analyzer and a real-time monitor/debugger for the ARTS Kernel which is being built for a distributed real-time testbed. The schedulability analyzer, called Scheduler 1-2-3, is a X11 window-based interactive tool and can verify whether the given real-time tasks can meet their deadlines under a specific scheduling policy. An Advanced Real-Time Monitor, ARM, also runs on X11 window and visualizes the target system's runtime behavior in real-time. ARM can also be used with Scheduler 1-2-3 to analyze simulated runs and its monitorability.

1. Introduction

Advances in Software Engineering provided us with a set of modern programming tools for building a large complex software system. Various tool sets can cover the wide range of the software development life cycle from a requirement analysis phase to debugging and maintenance phases [1, 4]. However, we cannot simply reuse the existing tool set for designing and building complex real-time computing systems without considering the time management capabilities. For instance, an interactive debugging tool in Smalltalk-80 [6] allows us to track down a "logical" bug in a program very well. However, it is almost impossible to detect or fix a "timing" bug for real-time programs [5]. A timing problem further complicates system modification, reconfiguration and maintenance [12].

Our attempt is to build an integrated tool set which can reduce the complexity of development of real-time software. In particular, we aim to detect and remove the potential "timing bugs" at the system design phase rather than at the system testing and debugging phase. For real-time software development, we recognize useful tools such as a "timing tool", "schedulability analyzer", "real-time monitor" and "real-time debugger". A timing tool should be able to estimate or measure the timing characteristics such as the average, worst, best case of the given software module, routine, or a system function. A schedulability analyzer can analyze or verify the schedulability of the given real-time task set under a specific scheduling policy. A real-time monitor can monitor and verify whether the system meets timing requirements at runtime. Finally, a real-time debugger should be able to capture unseen timing bugs at runtime with the minimum system interference.

However, there are many problems in building and integrating these real-time programming tools. For instance, the predictability of the analyzer heavily depends on what scheduling policies the target system uses and what types of real-time tasks it has. It is also necessary to take into account for various types of system overheads for the target system. A real-time monitor/debugger not only interferes with the processor cycles, but also every execution sequence of tasks and communications will be affected. These activities must be able to guarantee the minimum interference to the application software. The lack of operating system support often forces us to use ad hoc solutions for real-time monitoring/debugging problems.

In this paper, we describe an integrated real-time tool set for the ARTS Kernel which has been built for a network of SUN3 workstations. The tool set consists of an enhanced, X11 window [13] version of the schedulability analyzer, called Scheduler 1-2-3 [19] and the Real-Time Monitor [20] for the ARTS Kernel. In the following sections, we will first describe the functionality and the internal structure of Scheduler 1-2-3 and ARM (Advanced Real-Time Monitor/Debugger). Scheduler 1-2-3 is a X11 window-based interactive schedulability analyzer for creating, manipulating and analyzing real-time task sets. ARM is also a X11 window-based tool designed to analyze the target system's runtime behavior in real time. We then describe

---

This research was supported in part by the U.S. Naval Ocean Systems Center under contract number N66001-87-C-0155, by the Office of Naval Research under contract number N00014-84-K-073, and by the Federal Systems Division of IBM Corporation under University Agreement YA-278067. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of NOSC, ONR, IBM, or the U.S. Government.

---

1Sun3 is a trademark of Sun Microsystems, Inc.

2Of course, the name of Scheduler 1-2-3 came from Lotus1-2-3 which is a registered trade mark of Lotus Development Corporation.
the ARTS Kernel's built-in supports for object-oriented computational model, time encapsulation mechanism, and monitoring/debugging, and its performance. A comparison of our approach and other related work is also presented.

2. Real-Time Tool Set

The goal of our real-time tool set is to reduce the complexity in real-time system development. Timing is crucial for real-time systems. However, timing constraints have not been properly expressed in traditional programming languages or software development tools. A new tool set should be introduced. Figure 2-1 shows the system flow diagram in the ART testbed. Timing information on the target system is obtained by a timing analysis tool or given by a system specification. Scheduler 1-2-3[19] uses the given timing information, verifies the schedulability and synthesizes a part of a test program, which will be downloaded onto the ARTS Kernel. The Advanced Real-Time Monitor/Debugger (ARM), the enhanced version of the ART Real-Time Monitor [20], provides us with feedback information by visualizing the system behavior in real time. A timing analysis tool will be eventually integrated with other tools. In this section, we introduce Scheduler 1-2-3 and the Advanced Real-Time Monitor/Debugger for the ARTS Kernel.

2.1. Scheduler 1-2-3: A Schedulability Analyzer

The importance of Scheduler 1-2-3 is to foresee the schedulability of the target system at the design phase or whenever any subtle modifications of timing requirements of the tasks are necessary. By the schedulability analysis, we mean the analysis which analyzes whether or not a given real-time task set can meet its timing constraints under a given scheduling policy.

2.1.1. Objectives

The main objective is to provide an interactive design tool for complex real-time system development. For the existing systems, the Scheduler 1-2-3 can be used to predict the timing effects due to the software and hardware modification. Another objective is to utilize Scheduler 1-2-3 as a synthetic workload generator, which can be integrated with the other test tools, the timing tool and the real-time monitor/debugger. The objectives are summarized as following.

Schedulability Analysis

The schedulability is verified for the given hard deadline task sets under arbitrary scheduling algorithms, such as the rate monotonic[9], the first-in first-out (FIFO), the earliest deadline first and so on.

Response Time Analysis for Aperiodic Tasks

The performance of the given sets of aperiodic tasks with soft deadlines is given under arbitrary scheduling algorithms, such as Polling, Background and the Deferrable Server [7].

Easy-to-Use Interface

An interactive user interface is provided on a window system of bitmap display for ease of use. The user can enjoy the schedulability analysis through the user interface without precise knowledge of internal analysis procedure.

Synthetic Workload Generator

Scheduler 1-2-3 outputs a workload table. To confirm the schedulability of the given task set on a practical environment, a synthetic task set is generated based on the workload table and tested in the ART Real-Time testbed as illustrated in Figure 2-1.

2.1.2. Internal Structure

Scheduler 1-2-3 consists of four major modules. They are the Analysis Module, the File Interface Module, the User Interface Module and the Network Interface Module, as depicted in Figure 2-2. Those major modules are carefully designed for the ease of porting Scheduler 1-2-3 to other window systems or applying the analysis module to another purpose, such as a communication scheduling. In fact, The preliminary implementation had been done on the Sun3 workstation's window system. It currently has been ported to X11 window system.
The monitorability analysis will be discussed in 2.2.3.

File Interface Module

Generating a synthetic workload table and reading/writing a task definition file are done by this module. This module makes Scheduler 1-2-3 an efficient tool integrated with other real-time tools. The workload table would be included into a test program that puts synthetically generated workload to the ARTS Kernel, then the schedulability of the given task set is practically tested in the ART Real-Time testbed, which would be a piece of a C program. A task set definition table is a collection of binary data of the parameters of the given task set used to save the time to re-input the task set definition.

Network Interface Module

The simulator not only analyzes the schedulability but also generates a sequence of scheduling events, an event stream. This module sends the event stream over the network under 4.2 bsd UNIX\(^3\). This event stream has a compatible data structure with the one which is generated by the ARTS Kernel's scheduler object. ARM, therefore, can receive the events from either Scheduler 1-2-3 or the ARTS Kernel, and visualize them.

2.2. ARM: Advanced Real-Time Monitor/Debugger

Because a real-time system is time critical, the monitoring activity must produce a minimum amount of interference. Monitoring behavior might reduce the performance, and might introduce unexpected errors into the system. Our approach is one of software approaches which predicts in advance whether the given task set can meet its time constraint with the overhead produced by the built-in monitoring activity. Software approach is cheap but may severely degrade the performance. Of course, it is difficult to totally eliminate the interference with a software approach. Hardware approach, on the other hand, could achieve high degree of noninvasive monitoring, however, large amount of additional hardware is necessary and could be expensive.

2.2.1. Objectives

The objective of ARM is to "visualize" the system's runtime behavior for the designers of the ARTS Kernel and to make the monitoring/debugging overhead predictable. It also provides the user with a handle for remote debugging. The objectives are summarized below.

Visualization

The monitor should be able to visualize the system activity at an arbitrary level of abstraction. However, as a first implementation, ARM can visualize the system's scheduling decisions, namely context switchings, among periodic and

---

\(^3\)UNIX is a registered trademark of AT&T Bell Laboratory.
Using Scheduler 1-2-3, schedulability of the INS task set is being verified. In this example, in addition to the original INS tasks, an extra periodic task, called Reporter64 is created for real-time monitoring. Reporter64 can report 64 monitoring events per a message to the ARM's Visualizer.

aperiodic processes by means of a execution history diagram. In reminder of this paper, we use "event" to represent the combination of these scheduling decisions.

Monitorability Analysis
Performance degradation due to the monitoring/debugging features can be minimized and estimated before hand in our approach, although degradation cannot be avoided. Our approach uses the monitorability analysis. By The monitorability analysis, we can predict the maximum overhead caused by monitoring/debugging activities.

Remote Debugging
The eventual goal of ARM is to incorporate remote debugging functions, which will be performed by invoking the ARTS Kernel objects. The kernel support for debugging will be described in the next section.

2.2.2. Internal Structure
ARM is divided into three functional units; the Event Tap, the Reporter, and the Visualizer as illustrated in Figure 2-4. The Event Tap is a probe embedded inside the kernel to pick up the raw data on interesting events. The Reporter is in charge of sending the raw data to the Visualizer on a remote host. The Visualizer analyzes and visualizes the events which are sent from the Reporter. The Visualizer is also carefully designed for the ease of porting. The new version of the Visualizer has been ported to X11.

The Event Tap
It is part of the scheduler object, because we are currently interested in real-time scheduling policies. The Event Tap collects the scheduling related events and store them in private data buffer of the scheduler object. To minimize and predict the monitoring overhead, the Event Tap is not responsible for transferring the events over the real-time network.

"Process" and "task" are used interchangeably in this paper.
The Reporter

This is a process on the target host which is responsible for communicating with the Visualizer on remote host. The Reporter will periodically invoke an operation of the scheduler object. The operation extracts events stored in the object, packetizes them into an event message and sends it to the Visualizer.

The Visualizer

The Visualizer analyzes and visualizes the system behavior. It also provides a handle for remote debugging on a host outside of the target system. Events sent from the Reporter are interpreted to a visualized execution diagram. Not only visualizing the system behavior, the Visualizer provides the following analysis functionalities.

- The important statistics are calculated, such as total CPU utilization for periodic and aperiodic tasks, the number of successful completions and missed deadlines by abortion or by cancellation. The number of events per second is also reported.
- The Visualizer is capable of replaying an execution diagram for later analysis. An execution diagram can be played back, zoomed in and zoomed out. The statistics are also provided while a replay is going on. Figure 2-5 shows the Visualizer is monitoring and recording the system behavior.
- The functionality of timing analysis is also provided. The Visualizer can report the exact execution time of each process.

2.2.3. Monitorability Analysis

In real-time monitoring, it is very important to predict the maximum interference and capability of the monitoring process itself. Our approach is to predict the interference and the monitorability during the system analysis phase by using Scheduler 1-2.3. The Reporter process is added to the given task set as a periodic process so that the interference by the monitoring activities can be taken account into the schedulability analysis. The monitorability is also analyzed by using the worst case analysis. By the monitorability, we mean that the Reporter can keep reporting the system behavior to the Visualizer. In a sense of best effort, even if the monitorability is not guaranteed, the system behavior can be reported to certain extent. Nonetheless, events generated by the system will eventually overwhelm the capacity of the Reporter, and important events might be lost.

For the monitorability analysis, we should know how many events can occur in the worst case. It is assumed that there are \( n \) periodic tasks with periods \( T_i \) for \( 1 \leq i \leq n \). In accordance with Leinbaugh's analysis [8], task \( k \) with period \( T_k \) can be preempted \( \left\lfloor \frac{T_k}{T_j} \right\rfloor + 1 \) times in a period by task \( j \) with period \( T_j \). In general, the number of preemptions in the worst case is estimated in the following manner. However, in case of monitoring procedure calls or object invocation, it is impossible to predict how many events will incur in the worst case. A dynamic flow analysis scheme is necessary.

\[
\text{Total_preemption} = \sum_{i=1}^{n} \text{Preemption}_i
\]

\[
\text{Preemption}_i = \sum_{j=1}^{n} \left( \frac{T_i}{T_j} + 1 \right)
\]

where \( i \neq j \), and task \( j \) may arrive during an execution of task \( i \).

Meanwhile, under the rate monotonic scheduling, the number of preemptions can decrease. The rate monotonic scheduling defines that a more frequently executed periodic task is given a higher priority in execution than a less frequently executed task, and a task can preempt over another task with lower priority. Namely, task \( i \) can preempt task \( j \) if \( T_i < T_j \). Assuming the idle task whose period is infinitely long, every task arrival can be regarded as a preemption over the idle task. Let \( E_{\text{switch}} \) and \( T_m \) be the number of events caused by a preemption and the period of the monitoring task respectively, \( E_{\text{p}} \) the maximum number of events produced in \( T_m \) is calculated with the following expression.

\[
E_{\text{p}} = E_{\text{switch}} + E_{\text{m}}
\]
If $E_p$ is bigger than the number of events that can be reported in one period, the system behavior cannot be followed up by the Reporter. Examples of the monitory analysis are depicted in Figure 2-6 and 2-7. In Figure 2-6, it is reported that the system behavior can be monitored. On the contrary, in case of Figure 2-7, the number of events exceeds the Reporter's capacity.

**Figure 2-6: The Monitorability Analysis**

The Visualizer can depict the run-time system behavior from the event stream sent by the ARM's Reporter. The top six boxes indicate the command buttons. The top half portion of the tasks indicated by Lid 1 through 8 corresponds to the periodic activities defined in the INS task set and the bottom half is corresponding to the aperiodic activities. Character 'E' shown in the screen image indicates a periodic task terminates with its deadline being met; otherwise, 'A' or 'C' indicating the task is aborted or canceled due to missed deadline would be displayed. The bottom window can show various messages from the ARM's event analysis routine. Finally, the box at the center indicates a region which we can zoom in.

**Figure 2-5: The Visualized Runtime System Behavior**

Advanced Real-Time Monitor (ARM) version 4.2

Record Reset Zoom Dump Analyse Quit

Idle

Lid 1 Lid 2 Lid 3 Lid 4 Lid 5 Lid 6 Lid 7 Lid 8 Lid 9 Lid 10 Lid 11 Lid 12

Time: 25000 ms.
Periodic tasks: 8
Aperiodic tasks: 0
Total CPU utilization: 1.000 (cyclic tasks (0.912) (acyclic tasks (0.088))
Met deadline: 171
Missed deadline: 0
Aborted: 0
Canceled: 0
Events: 147 events - 57.000 per second
to utilize the schedulability analysis scheme under several time driven scheduling algorithms. Meanwhile

2.3. Future Work

One of our primary objectives is to provide an integrated tool set for real-time programming. Scheduler1-2-3 helps us to utilize the schedulability analysis scheme under several time driven scheduling algorithms. Meanwhile ARM is very useful in both debugging the ARTS kernel and evaluating a scheduling algorithm.

An important extension planned is to provide more sophisticated capabilities. For instance, is case of missed deadline, the diagnostics message should include intelligent suggestions (or a result from a period transformation method) to improve the schedulability. The current analysis for the aperiodic task set is to estimate their response time under a given scheduling policy. However, a reversed analysis is also important. That is, for a given response time requirement of the aperiodic task set, Scheduler1-2-3 should be able to estimate the necessary specifications of the deferrable server or anything that can produce the given response time. The goal of this reversed analysis would be to exploit a scheduling scheme for a mixture of periodic and sporadic tasks. Finally, we are also planning to extend the domain of the real-time computational model to a distributed environment.

Regarding ARM, remote debugging facilities should be implemented so that necessary information can be obtained by stimulating the system remotely. These facilities should have a support by the underlying kernel. The ARTS kernel supports debugging primitives which allow remote debugging on an object-oriented computing model. The kernel support will be described in the following section.

3. ARTS Kernel Support

The ARTS Kernel provides a distributed real-time computing environment based on an object-oriented model. In the ARTS, every computational entity is represented as an object, called an "artobject", and it can be a "passive" as well as "active" object which can contain more than one (lightweight) process. In this section, we describe our object model, the kernel mechanisms which can confine timing errors, and a part of the ARTS Kernel interface which is related to system monitoring and debugging support at various levels of abstraction.

3.1. Object Model

We view an "artobject" as a basic module for embodying a distributed abstract data type. An artobject is a distributed abstract data type consisting of two principle parts: a specification and a body. The artobject specification, describing the external user's view of the artobject, consists of a set of operations which other artobjects use to activate services offered by the artobject. The artobject body consists of a set of "passive" procedures or contains at least one process, called INITIAL, as well as other processes which share a set of shared data objects. We refer to the former type as a "passive" object and the latter one as an "active" object.

When an instance of the active artobject is created, the INITIAL process is created and run immediately. We implemented these processes in the artobject as lightweight processes for which creation and destruction can be done cheaply. The shared data objects in an artobject are totally hidden from other artobjects and may be accessed by other artobjects only by invoking an operation defined in the specification part.

Interaction among artobjects is performed by operation invocation. An invocation request to a passive artobject is performed as a remote procedure call, while a request to an active artobject is performed by an explicit "Request-Accept-Reply" sequence. Namely, an artobject invokes an operation by sending an invocation request message to the destination artobject explicitly, and then it waits for a result. When the destination sends a result back, the caller wakes up and receives the result. From a user's point of view, there is a common invocation syntax, like "object.opr(args)", however the ARTS Kernel provides more flexible invocation mechanism which can perform an asynchronous request and one-to-many communication [3, 16].

The skeleton of a simple artobject is shown in Figure 3-1.
... Reply(Req, TrasId, result);
end INITIAL;

operation opr1(arg, ...) => (result)
  within time except recovery-opr1(arg, ...)
operation recovery-opr1(arg, ...) => (result)
operation opr2(arg, ...) => (result)
  within time except recovery-opr2(arg, ...)
operation recovery-opr2(arg, ...) => (result)
... end Sample;

Figure 3-1: A Skeleton of Artobject Declaration

For each operation definition of an artobject, the designer of the object must provide a "time fence" (i.e., worst case time bound) and its time exception handling routine by specifying "within time except recovery-opr()". The "time" indicates that the operation must be completed within the "time limit"; otherwise the specified "recovery" operation will be executed. In the timing exception routine, it may cause a critical section problem within the requested operation. The basic principle in the ARTS is that the designer must put the state of any critical data object "back" or "forward" to a consistent state. In real-time applications, we often prefer to use the "forward" recovery scheme (i.e., we call it a "compensation" routine [17], since an "undo" operation cannot be performed against any external (real) state change.

3.2. Time Encapsulation Support

The notion of "time encapsulation" is to encapsulate (or confine) each module's timing error within the module. This also requires us to specify a timing requirement explicitly for each object. The ARTS Kernel supports the time encapsulation among real-time objects/processes by providing two mechanisms: the integrated "time-driven" scheduler and "time fence" mechanism.

The integrated time-driven scheduler (ITDS) uses a rate monotonic scheduling policy for periodic "hard" real-time tasks and adopts a value function based scheduling policy [10, 18] with the "deferrable server" for "soft" aperiodic tasks. In the ITDS scheduler, we first analyze the given "hard" real-time task set's schedulability, then we can compute the maximum CPU utilization that the deferrable server can consume. In other words, we try to provide the maximum amount of CPU cycles to the aperiodic task set while we can guarantee meeting all hard real-time tasks' deadlines. In summary, the ITDS scheduler can guarantee the following:

- Schedulability of the "hard" periodic tasks (together with the result from Scheduler1-2-3),
- Value function based "soft" real-time task scheduling, and
- Overload control based on the given value functions of the aperiodic tasks.

The "time fence" is a mechanism to detect a timing error at every object invocation at runtime in the ARTS Kernel. Since we must specify the worst case timing requirement for each operation in an object, the kernel can perform the timing check. This is in a sense similar to the array boundary check.

Suppose that an artobject A's process P invoked an operation X on artobject Q (i.e., "Q.X()"), then the ARTS Kernel will initiate the following timing check during the invocation protocol. Suppose Pct and Pwst stand for P's current time and P's worst case slack time respectively. And P's current fence time (i.e., P's starting time plus the current fence value) is represented by Pfcf:

- When P invokes Q.X(), P's request message will carry Pct as well as Pwst (i.e., Pwst = Pcf - Pct) in P's local time.
- After the request message was received by the communication manager at Q's site, it will check whether P can meet the current fence value Qcfv by the following condition:

  \[
  \text{art_error("time fence violation");}
  \]

3.3. Monitoring/Debugging Support

The objective of the monitoring and debugging support primitives in the ARTS Kernel is to reduce the complexity of application development in a distributed real-time environment. The primitives are useful not only for building a debugger, but also providing application specific monitoring functions. In ARTS, there are three sets of basic primitives for any type of objects/processes:

- "Freeze/Unfreeze" object or process's activities,
- "Fetch/Store" object or process's data object, and
- "Watch/Capture" object or process's communication activities.

The "Freeze/Unfreeze" operations control the activity of an artobject or process by stopping or resuming it. The "Fetch/Store" operations can be used for retrieving and storing data object from a specific artobject or process. The "Watch/Capture" operations monitor the communication activity among artobjects by coping or intercepting the selective messages. In the following, we will briefly describe the functionality of the kernel primitives.

3.3.1. Freeze and Unfreeze

The Object_Freeze primitive stops the execution of an artobject (i.e., all its associated processes), while a Process_Freeze primitive halts a specific process for inspection. An Object_Unfreeze and Process_Unfreeze primitive resumes a suspended artobject and process respectively. While a process is in a frozen state, many of the factors used for making scheduling decisions can be selec-
tively ignored. For instance, a timeout value will be ignored by specifying a proper flag in the option field. In similar context, the ARTS Kernel also provides "Node_Freeze" and "Node_Unfreeze" primitives to halt or restart all of the client's activities in a specific node. The Freeze/Unfreeze primitives are defined as follows.

\[
\begin{align*}
\text{val} &= \text{Object_Freeze}(\text{oid}, \text{options}) \\
\text{val} &= \text{Object_Unfreeze}(\text{oid}, \text{options}) \\
\text{val} &= \text{Process_Freeze}(\text{pid}, \text{options}) \\
\text{val} &= \text{Process_Unfreeze}(\text{pid}, \text{options}) \\
\text{val} &= \text{Node_Freeze}(\text{nid}, \text{options}) \\
\text{val} &= \text{Node_Unfreeze}(\text{nid}, \text{options})
\end{align*}
\]

Note that the "options" parameter can be used to selectively ignore on-going target activities. For instance, it can effectively skip the timeout processing of the target activity while the target object/process is frozen.

3.3.2. Fetch and Store

A Fetch primitive inspects the status of a "running" or "frozen" artobject or process in terms of a set of values of data objects. The specific state of the artobject or process will be selected by a data object-id. The state includes not only the status of private variables, but also includes object/process control information. The Fetch/Store primitives are defined as follows.

\[
\begin{align*}
\text{fval} &= \text{Object_Fetch}(\text{oid}, \text{dataoid}, \text{buffer}, \text{size}) \\
\text{sval} &= \text{Object_Store}(\text{oid}, \text{dataoid}, \text{buffer}, \text{size}) \\
\text{fval} &= \text{Process_Fetch}(\text{pid}, \text{dataoid}, \text{buffer}, \text{size}) \\
\text{sval} &= \text{Process_Store}(\text{pid}, \text{dataoid}, \text{buffer}, \text{size})
\end{align*}
\]

Note that "dataoid" indicates a private data object's id within a given object or process. The system do not guarantee the consistency of the target data unless the target was frozen in the consistent state.

3.3.3. Capture and Watch

The Capture primitives capture on-going communication messages from the specified artobject or process. A Object_Capture primitive captures all incoming requests as well as outgoing reply messages to a specified artobject, and can also select a target message based on the name of the operation. A Process_Capture primitive captures all incoming messages and outgoing reply messages for a specified process. The Watch primitives are similar to the Capture primitives except that all monitored messages are duplicated, not captured. The "Capture/Watch" primitives are defined as follows.

\[
\begin{align*}
\text{val} &= \text{Object_Capture}(\text{oid}, \text{commttype}, \text{opr}, \text{requestor}) \\
\text{val} &= \text{Object_Watch}(\text{oid}, \text{commttype}, \text{opr}, \text{requestor}) \\
\text{val} &= \text{Process_Capture}(\text{pid}, \text{opr}) \\
\text{val} &= \text{Process_Watch}(\text{pid}, \text{opr})
\end{align*}
\]

Note that "commttype" argument indicates either "IN" or "OUT" type and "IN" selects an incoming invocation request message and "OUT" indicates an outgoing reply message. The "requestor" specifies the requester's oid or "ANY_OID". Similarly, the "opr" parameter can be a specific operation name or "ANY_OPR".

3.4. The ARTS Kernel and Monitoring Performance

The current implementation of the ARTS Kernel and ARM have been done on a network of SUN3 workstations. The Reporter process runs on the ARTS Kernel, however, the Visualizer on top of X11 window system on a UNIX 4.2bsd host machine. The basic performance of the ARTS Kernel and the real-time monitor can be summarized as follows:

| Event Logging Overhead | 3 msec per message (64 events) |
| Event Record Size     | 20 bytes                        |
| Event Message         | 64 events per message           |
| Event Message Transfer Speed | ~40 msec per event |
| Visualizer's Message Receiving/Display Speed | ~150 msec per message (64 events) |

4. Related Work

We introduced our approach to build an integrated tool set which can reduce the complexity of real-time software. Tools included in the integrated tool set can not only operate in stand alone manner but also be cooperative with other tools and the ARTS kernel. Many different algorithms and their implementations have been proposed to solve the complexity of real-time system development; however, there have been few attempts to introduce an integrated tool set. In this section, we will take a look at some related work to sketch out the characteristics of our approach.

A well known formula to analyze schedulability of periodic tasks has been given by Leinbaugh [8]. This approach assumes that a task consists of critical sections, use of devices and non-critical sections. Then, it provides us with a theoretical schedulability analysis with critical sections and use of device. Our approach has not yet integrated the consideration for critical sections. His approach also takes advantage of being independent of a certain scheduling policy. Subtracting critical sections, device usage time and blockage time by preemptions from a period, the ratio of cpu time allocated to that task's non-critical sections is estimated. Then the summation of the ratio of all tasks is verified. If the summation is less or equal to 100 %, that task set is schedulable.

A new real-time programming language Real-Time Euclid [14] was designed for real-time programming. This language comes up with a set of schedulability analysis provisions built-in for the sake of schedulability analysis, which
collects timing and task dependency information from the target programs. The schedulability analyzer based on Real-Time Euclid, called Schedalyzer, uses a schedulability analysis algorithm that is basically identical with the Leinbaugh's formula, all tasks are scheduled the closest-deadline-first. The strong points of this approach come from its "language-oriented" scheme. It is unnecessary to depend on a timing tool which our approach still lacks. It is expected to collect exact timing information. And the language can be more suitable for real-time programming than traditional high level languages. This language features static data structures, time-bounded loops and so on. On the other hand, Real-Time Euclid has a limitation due to its "language-oriented" approach, which has a drawback that other systems which are not written in Real-Time Euclid cannot be analyzed.

The necessity of continuous system monitoring was addressed by Svobodova [15]. She employs the abstract data model. In her model, the monitoring activity can be specified as an operation of each abstracted data. If an interesting event occurs to that abstract data, a monitoring signal is propagated to the monitor. This approach is very similar to ours, whereas this model has a richer set of monitoring operations in a sense that every operation of every object can be monitored while only scheduling activities can be monitored in our approach. The advantage of our approach is that the maximum interference caused by the monitoring can be bounded with the monitorablity analysis. Of course, when interesting events are extended to object invocation or procedure calls, we also have to face the problem of unbounded numbers of events.

IPS [11] models the program hierarchy with five levels, namely whole program level, machine (host) level, process level, procedure level and primitive activity level. The lowest level consists of primitive activities. In IPS, those primitive activities are detected by the scheduler, namely blocking, unblocking and so on, and then they will be mapped into higher levels of the program hierarchy. For example, a message send event could be mapped into the total message traffic in the program, or it may be mapped to the message traffic between hosts. This approach makes it possible to monitor the system behavior in multi-level abstraction, while our approach is limited to single level monitoring, namely task scheduling level.

5. Summary

In real-time systems, nasty "timing bugs" are very difficult to capture and eliminate from the system. Our approach was to provide a good set of real-time software tools and operating system support so that we can debug the timing error at the early stage of the design phase rather than the software testing phase. In this way, we will be able to reduce the complexity of the system design and major maintenance cost of the real-time software.

In this paper, we described the integrated real-time tool set developed for the ARTS Kernel. We first described and demonstrated Scheduler 1-2-3 which can analyze the schedulability of the task set under a given scheduling policy. Scheduler 1-2-3 can also generate a synthetic workload for the ART testbed or can produce its execution history diagram together with ARM. It then allows us to verify our scheduling policy on the ART testbed.

We also discussed the architecture of ARM which based on a "software approach" for monitoring. It is clear that our approach is not a non-intrusive monitoring/debugging scheme. However, we demonstrated that the monitoring schema is very practical, yet predictable when the real-time monitor is used with the integrated time-driven scheduler. Furthermore, the separation between the Reporter and the Visualizer allows us to use the monitor for many embedded system. This separation also allow us to extend the monitor to visualize many different levels of activities, such as programs, objects, procedures, statements, and variables. Finally, we also discussed the ARTS Kernel support for time encapsulation mechanism and real-time monitoring/debugging activities.

Acknowledgement

We would like to thank John Lehoczky, Lui Sha, and Ragunathan Rajkumar for their contributions to the analysis algorithms for Scheduler 1-2-3. We also thank Bruce Miller for the development of X11 version of both tools and Cliff Mercer for his valuable comments.

References

[9] Liu, C. L. and Layland, J. W.  
Scheduling Algorithm for Multiprogramming in a Hard Real-Time Environment.  


IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs.  
In Proc. 7th Int. Conf. on Distributed Computing Systems.  

[12] Mok, A. K.  

X Window System Protocol.  
Massachusetts Institute of Technology, 1987

[14] Stoyenko, A. D.  
A Schedulability Analyzer for Real-Time Euclid.  

[15] Svobodova, L.  

Shoshin OS: a Message-based Operating System for a Distributed Software Testbed.  
In Proc. of 16th Hawaii Int. Conf. on System Science Volume 1, January, 1983.

[17] Tokuda, H.  


[19] Tokuda, H. and Kotera, M.  
Oct., 1988

A Real-Time Monitor for a Distributed Real-Time Operating System.  