Real-Time Systems Design in Ptolemy II: A Time-Triggered Approach

N. Vinay Krishnan

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Prof. Edward A. Lee

Research Advisor

* * * * * *

Prof. Thomas A. Henzinger

Second Reader

Memorandum No. UCB/ERL M04/22

12 July 2004

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

In this report is described a software infrastructure to enable users to design hard real-time systems from Ptolemy II [1]. The Giotto [2] domain within the Ptolemy II design environment is made use of to model systems which are then compiled and executed on KURT-Linux [3], a real time flavor of Linux.

The first stage of the software takes a graphical model as an input to generate intermediate code in the C language. This intermediate code consists of the task-code to be executed, as well as a representation of their timing requirements.

The second stage, called the Embedded Machine [5] reads in the timing information and interprets it to release the tasks for execution as per the stated requirements. The released tasks can either be assigned to a standard scheduler such as EDF, or to a scheduling interpreter called the Scheduling machine, or S Machine.

The S Machine was developed to gain fine grained control over the scheduling of tasks. The S Machine requires as input scheduling information that specifies a time line for the tasks involved thus giving the designer maximum flexibility over task scheduling, and consequently greater resource utilization.

The E&S Machines when compiled along with the generated task and timing code for the KURT-Linux platform forms an executable that delivers predictable real-time performance. The benefit this approach offers is that the real-time tasks can run along with ordinary Linux tasks without the timing properties of the real-time tasks being affected.

An audio application was designed to illustrate the effectiveness of this tool-flow, which achieved a timing uncertainty of less than ±30 microseconds in its task execution times.

Acknowledgements

First and foremost I would like to thank Prof. Edward Lee for all the help and support he has offered me. From taking me into his research group, finding out my interests and letting me explore that, to the numerous discussions where he gently brought me back on to the path that I, in my infinite wisdom, was contentedly straying from, to the care with which he reviewed my report and offered suggestions, it would not be wrong to say that without his guidance I would have still been floundering in the oceans of academia. Thank you.

I would also like to thank Prof. Thomas Henzinger for agreeing to take time out to be my second reader in spite of all his new responsibilities, and his amazingly hectic schedule.

Numerous people helped me complete this project by sharing their knowledge and offering guidance. I do not dare try to order their names as all of them deserve to be the first. However the serial nature of the written word forbids me from mentioning them all at once. Therefore, in no particular order, I would like to thank Prof. Douglas Niehaus of Kansas University and his students Michael Frisbie, Tejasvini, and all the others who helped me understand the intricacies of KURT-Linux, actually create applications which work, and helped debug issues with DSUI. Christoph Kirsch and Marco Sanvido were very helpful with explanations of the Giotto semantics and the E Machine. Elaine helped me out with the elusive installations of the ALSA driver on the KURT platform. Steve of the Ptolemy group offered very valuable suggestions as to the direction of the project when it was at crossroads. Yang and Xiaojun also deserve a special mention for tolerating my many intrusions into their cubicle with innumerable questions about the workings of Ptolemy, and patiently answering them.

Thanks also go out to Susan Gardner for taking care of me during my stint in the third floor lab, and for lending a sympathetic ear to narrations of all my trials and tribulations.

A big thank you to all these people and apologies to those who my myopic memory failed to bring into focus.

This project was funded by NSF cooperative agreement no. CCR-0225610

Table of Contents

1. Introduction

The design of predictable real-time systems is an important field today due to the increasing employment of embedded systems in the time critical aspects of modern-day devices. Tools that help in the design of such systems are therefore growing in importance. The Ptolemy II [1] project addresses this and offers a modeling and simulation environment for real-time systems by including among its models of computation, the Giotto [2] model of computation. This project looks at extending the features of Ptolemy II from the simulation of real-time systems to the actual generation of executable code that exhibits the same properties that the initial model does. The goal is to develop a tool-chain that takes as its input a Ptolemy II graphical model of a real-time system and outputs a software application that has predictable performance.

In designing the software, the results of three project efforts were incorporated and built upon, viz. Ptolemy II [1], Giotto [2], and KURT-Linux [3]. In the following section, an introduction to each of them is given with relevant concepts from the original papers being reproduced for the sake of self-containment. Section 2 describes the design of the software and specifies the algorithms used. Section 2.3 contains a description of how one might use the software by giving an example. An actual audio application built using the software is discussed in section 3. Section 4 lists some of the limitations of the software, and section 6 contains possible future extensions to the project.

1.1. Background

1.1.1 Giotto

Giotto "provides an abstract programmer's model for the implementation of embedded control systems with hard real-time constraints" [2]. The semantics it offers to express system properties are in close alignment with a mathematical model of a control system. This is desirable since control systems are often designed at a mathematical level of abstraction to facilitate effective optimization of control laws, and validate the performance and functionality of the model. The onus of translating a mathematical control model into software falls to the developer who has to create software tasks corresponding to the computations and assign priorities so as to meet the requisite time

constraints on the specific hardware. This is an ambiguous process with the desired results achieved through the use of an iterative process of testing and debugging.

Giotto removes the ambiguity by providing the developer a formal, machine-readable language in which to specify the system requirements that is easily verifiable by the system designer. Giotto follows the time-triggered design paradigm, making it particularly suited for embedded control systems which have hard real time constraints. It requires the specification of the functionality and timing of the computations of a system. The mapping of these properties on to a particular platform is a compilation issue which is kept independent from the specification. This gives Giotto one of its advantages, namely the separation of the design concerns from the platform dependent execution details. This makes it attractive as a language of communication during the initial stages of the design process.

Once a software system is specified using Giotto, it remains to map this specification onto a hardware platform. This process can be largely automated through the use of compilers and virtual machines. One such method, which has been used in this project, is the *Embedded Machine* [5]. The compilation of Giotto guarantees the preservation of the functionality and timing, thereby producing an executable software application that remains faithful to the original mathematical model.

Any control application consists of a set of periodic computations, with elements being added and deleted to the set as the system progresses. Giotto has a basic functional unit of a *task* representing a single periodic computation which is a piece of code written in a programming language like C. *Modes* represent a fixed set of such tasks. A Giotto program representing a system is a set of modes. It can be in one mode at any point in time, repeatedly invoking the corresponding set of tasks concurrently. A task can be present in multiple modes, with different timing properties in each, and switching between modes is how the addition and deletion of tasks in the system is accomplished in a Giotto program.

Tasks communicate data through *ports*. Every task will have zero or more input ports from which it reads data upon being invoked, and feeds the result of its execution into zero or more output ports. Other than these task ports, a Giotto program also contains sensor ports which read values from sensors, and actuator ports. Sensors and actuators are

the means by which a Giotto program reads values from the external environment and conveys decisions to it. Every port is persistent in the sense that the port keeps its value over time until updated. Sensor ports are updated by the environment; all other ports are updated by the Giotto program. Task ports communicate data with each other, as well as with sensor and actuator ports, by *drivers*, which is code that transports and converts values between these ports.

While tasks are segments of computation dispatched by a Giotto program to execute on the hardware platform with a finite non-zero amount of time, drivers are considered by Giotto to be instantaneous bits of code. They are called synchronously by a Giotto program in between dispatching tasks. To quote from [2], it is assumed that "drivers satisfy the synchrony assumption [13], that they can be executed before the environment state changes".

As mentioned above, all the scheduled computations, the synchronous communications between ports, and mode switching occur in real time, a consequence of the time-triggered nature of a Giotto program. "These time-triggered semantics enable efficient reasoning about the timing behavior of a Giotto program, in particular, whether it conforms to the timing requirements of a mathematical (e.g., Matlab) model of the control design." [2].

An example time line for an invocation of a task *t* shown in Figure 1.

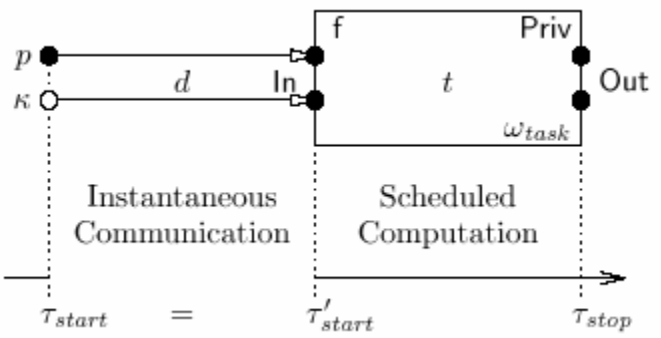

Figure 1: Time line for the invocation of a Giotto task *t* (reproduced from [2])

The invocation starts at some time τ_{start} with a communication phase in which the input-port values are loaded. As per Giotto semantics, the communication phase —i.e., the

execution of the input port driver d — is performed in logically zero time. This synchronous communication phase is followed by a scheduled computation phase where the task t is carried out. At time τ_{stop} the state and output ports of t are updated to the result of the task computation. The length of the interval between τ_{start} and τ_{stop} is determined by the frequency ω_{task}. This determines the number of times task t is executed over one Giotto period, specified in the program. Task t is said to be *logically running* from time τ_{start} to time τ_{stop}. The Giotto abstraction does not specify when the actual computation of f is performed between τ_{start} and τ_{stop}. However, the times at which the task output ports are updated are known, and therefore, "for any given real-time trace of sensor values, all values that are communicated between tasks and to the actuator ports are determined" [2]. Any Giotto realization has to be faithful to this abstraction; for example, task inputs may be loaded after time τ_{start}, and the execution of f may be preempted by other tasks, as long as at time τ_{stop} the values of the task output ports are those specified by the Giotto semantics.

A Giotto program can be realized on a particular program using a compiler and virtual machine such as the Embedded Machine, or E Machine [5]. The Embedded Machine is a virtual machine that aligns the real time interaction between software and physical processes. It separates the running of embedded programs into two phases. The first platform-independent phase interprets and runs E Code (code understood by the E Machine) which is compiled from Giotto programs by the E Machine compiler. E Code supervises the timing – not the scheduling – of application tasks. E Code is independent of any particular platform, and is therefore inherently portable. E Code also exhibits predictable timing and output behavior for a particular input behavior, making it a good choice to represent time-triggered Giotto programs. The generated E Code can be checked for schedulability using [6]. However, this requires knowledge specific to the platform such as the Worst Case Execution Times (WCET's) which can optionally be supplied as annotations in the Giotto code. Once the E Code determines when the tasks have to be released for execution in a Giotto period, they are passed onto the platform dependent phase of the E Machine.

The platform dependent phase takes care of executing the released tasks on the underlying hardware. The tasks can be scheduled by the E Machine using any standard scheduler such as EDF. However, more fine-grained control is achieved through the use of the S Machine.

The S Machine is a platform-dependent scheduling interpreter which interfaces with the E Machine. The tasks released by the E Machine are dispatched by the S Machine according to a schedule specified using a program called S Code. The S Code or Scheduling Code that is interpreted by the S Machine lays out the time line of the tasks to be executed. Due to the flexibility in scheduling policies it offers, it is inherently more expressive than any particular scheduler. It has been shown that using the S Machine results in an improvement in CPU utilization over the EDF scheduler [8]. In addition to providing the designer with flexibility over the task scheduling, the E Code and S Code can together be used to verify the schedulability of a set of tasks on a specific platform [7]. Since it is easier to verify the schedulability than to generate a feasible schedule, the E&S Codes, if distributed along with the tasks, relieve the user from having to come up with a feasible schedule. In essence they carry the proof of schedulability with them. In our design, the E & S Machine shall together be called the ES Machine. A further description of the ES Machine can be found in the Design section.

This project uses Giotto as the model for the real-time systems designed using its software. The graphical modeling and simulation properties of the Ptolemy II framework are exploited by using its Giotto domain as the initial design interface.

1.1.2 Ptolemy II

Ptolemy II is the current software incarnation of The Ptolemy Project [1]. This project being carried out at UC Berkeley “studies modeling, simulation, and synthesis of heterogeneous concurrent systems, with a focus on Embedded Systems”. Ptolemy II is a graphical design tool for hierarchical and component based system models. Modeling a system in Ptolemy II lets us take advantage of the many features it offers viz. a graphical user interface, type-checking, simulation of the design, and so on.

Ptolemy II follows the actor-oriented design approach. While the popular object-oriented approach stresses on the structure of a model or program by offering such features as modularity and hierarchy, the actor-oriented approach stresses concurrency

and correctness making it well suited for the domain of embedded systems. The basic unit of computation in Ptolemy is the *actor*, which executes and communicates with other actors forming a model of a system. Modularity is offered by viewing each actor as an individual entity offering a set of features. Well-defined interfaces dictate how the actor communicates with the rest of the model and abstract away the internal workings. An interface consists of *ports* which represent "points of communication" for an actor, and *parameters* which allow the designer to fine tune the execution.

A very powerful abstraction offered by Ptolemy is that of *hierarchy*. An actor may be an atomic unit of computation (atomic actor), but may also consist of other actors inside it, executing together to deliver some coherent functionality (composite actor). Such an actor may even contain a complete system model within it. To facilitate this, models also have ports and parameters distinct from those specific to any individual actor in a model. Such ports can be connected to any actor within the model. These model ports and parameters are connected to the ports and parameters of the actor abstracting this model.

Every Ptolemy II model has a *director*, which dictates how and when the actors fire and interact with each other, and consequently, how the system behaves. It does this by following a set of rules called the *model of computation* of the system. "The model of computation is a description of the behavior of a system. A model of computation may even have more than one set of rules, in that there might be distinct sets that impose identical constraints on behavior" [1].

Various models of computation exist which lend various properties to the system by virtue of the restrictions they impose. *Synchronous Data Flow* (SDF), and *Finite State Machines* (FSM) are two examples. The choice of a model of computation for a system can give or take away useful properties which might in turn affect the usefulness of the model for verification or simulation. For embedded systems, the most useful models of computation handle concurrency and time. This is because embedded systems consist typically of components that operate simultaneously and often interact with the simultaneous events in the physical world. In addition, the responses of these components to the stimuli are often required within a fixed time frame after which their usefulness

degrades. From this perspective, the semantics of Giotto are very appropriate as a model of computation for designing Embedded Systems.

The Giotto domain in Ptolemy II has actors representing the various tasks. The abstraction of hierarchy offers the actors the liberty of being models within themselves with any arbitrary model of computation. The concept of ports is similar in Giotto and Ptolemy, with input ports to the model representing sensors and output ports to the model representing actuators. The period of a Giotto iteration is specified using a parameter in the Giotto director, and frequency of the tasks are specified by parameters within the actors representing the tasks. Further details as to the implementation of this domain shall be discussed in the design section.

The Ptolemy II Giotto model can be compiled down to a Giotto program and C code required for the ES Machine to run. The platform upon which the ES Machine executes is a real-time flavor of Linux developed in Kansas University, KURT-Linux [3].

1.1.3 KURT-Linux

Linux as an operating system is gaining favor with the world because of its open-source nature and the high level of customization it offers. Linux has been entering the domain of embedded systems as processors become strong enough to support multithreaded operating systems. With the increasing need for real time operating systems in embedded devices, it wasn't long before real time versions of Linux started to appear. RTLinux [14] and KURT-Linux are two examples of such. KURT-Linux stands apart from RTLinux in the sense that RTLinux takes the "two executive" approach, i.e. RTLinux runs the regular Linux kernel as a virtual machine atop an RT Core, while KURT-Linux looks at modifying the Linux kernel to achieve real time performance for its tasks. The advantage of this is twofold.

Firstly, the two executive approach places restrictions on the sharing of data between real time and non-real time tasks. Special actions such as saving to a file, IPC or sending a network packet are required. While this might not be an issue for normal computations, it would create stumbling blocks when trying to create GUI's to display information on the status, or the results of real time task computations. Access to the non real time parts of the operating system was essential in the example audio application. The ALSA driver [11] for Linux was used as an interface between the real time tasks and

the sound card. The integration of the real and non-real sides of the operating system in KURT-Linux easily facilitated communication of data between the real-time application tasks and the non real-time ALSA API. Using RTLinux would have required the writing of special software for interfacing with the ALSA sound driver, or even writing a customized sound driver.

The second advantage is in footprint size. While numbers vary, the average size of the RTLinux kernel is measured in megabytes. KURT-Linux has a footprint in the 512KB area. The downside of KURT is that its performance numbers are below that of RTLinux. However, the results of some recent tests by the KURT-Linux group under Group Scheduling and the IRQ modifications indicate a performance envelope very close to that of RTLinux.

KURT-Linux achieves real time performance by primarily modifying the scheduling algorithm of the Linux kernel. Any and all tasks registered as real time tasks are offered priority over ordinary Linux tasks. Since interrupt requests can disrupt the normal scheduling routine, KURT-Linux has modifications built into the interrupt handlers to block interrupts pertaining to ordinary Linux tasks until the real time tasks have finished execution.

KURT-Linux also offers extensive profiling mechanisms in the guise of DSKI (Data Stream Kernel Interface) and DSUI (Data Stream User Interface) [12]. These are API's provided whereby a user can place hooks called instrumentations points in his code, that cause event logs with the timestamp to be recorded. DSKI is meant to be used in kernel programs and DSUI in user programs (applications). These low overhead data collection mechanisms proved extremely useful in measuring the timeliness of applications generated using the tool chain.

KURT offers a variety of scheduling options to the application designer, from *earliest deadline first* to *round robin* to *explicit plan*. In fact, the presence of the explicit plan scheduler was what made KURT an attractive choice for being the E Machine platform.

The explicit plan scheduler is one among the default schedulers offered as part of the group scheduler package in KURT-Linux. Group scheduling [9] is a hierarchical scheduling abstraction. It brings in the notion of grouping together tasks which are

similar, e.g. the processes of an application. A member of a group can be either a process or another group, thus bringing about the notion of hierarchy. Each group has a scheduling decision function associated with it.

The explicit plan scheduler is one such scheduling decision function. Rather than relying on priority based scheduling or strictly periodic schedules, KURT schedules are explicitly specified by the application programmer. When using the explicit plan scheduler, the application designer submits a plan laying out the time schedule of the application threads or processes in nanoseconds. This gives the designer very fine-grained control over the scheduling of tasks, which is what S Code requires. Given the features of the explicit plan scheduler, it was the scheduler of choice to write the S Code interpreter. A downside of harnessing the power of the explicit plan scheduler was that one of the requirements of S Code semantics had to be modified. This will be detailed in the design of the scheduling interpreter.

2. Design

The tool flow design is divided into two parts. The first part reads in a graphical Giotto model designed in Ptolemy II and generates code suitable for the ES Machine to read. This shall be called the ES Machine Code Generator.

The second part of the design consists of actually designing the ES Machine on the KURT-Linux platform which can then interpret the E Code and S Code generated by the first part and run the tasks according to the timing characteristics specified by this code.

The ES Machine shall be described first since the design of the framework code generator is dependent on the ES Machine design.

2.1. The ES Machine

2.1.1 Introduction

The detailed design of the E Machine is given in [5]. First a few basics necessary to understand the design of the E Machine are presented, followed by a description of how the KURT API was used to implement the interpreters.

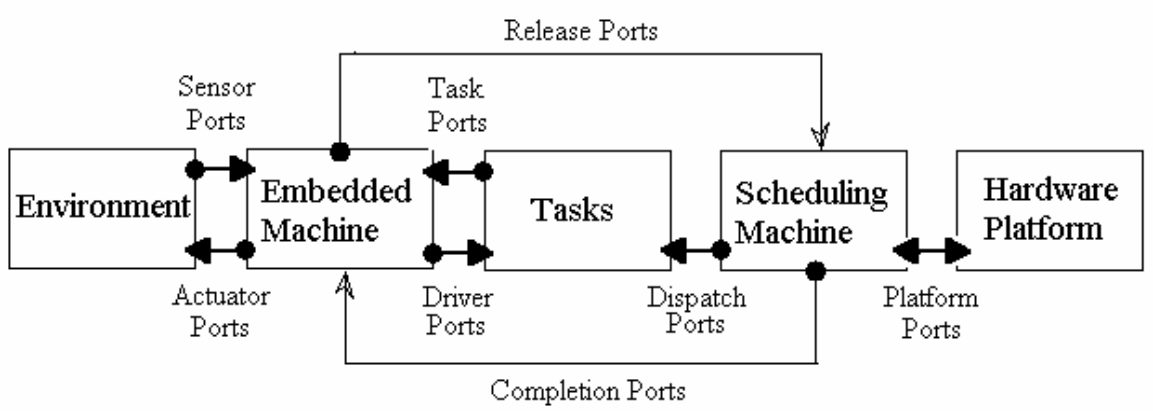

Figure 2: The E Machine and S Machine

The E Machine accepts inputs from two sources, the environment and the tasks. The environment feeds it inputs through *sensor ports* which can result in either the E Machine changing modes, i.e. changing the timing properties of the tasks it controls, or just forwarding the sensor values on to the tasks. The tasks communicate, through *task ports*, maintenance information and their outputs. Consequently, the task ports are also called the *output drivers* of the various tasks. The S Machine feeds information as to whether tasks have completed execution or not (signifying a deadline violation) through the *completion ports*.

The E Machine communicates to the environment through *actuator ports*. The intention is to convey data to actuators that carry out tasks external to the E Machine system. *Driver ports* are used to pass on output data received through task ports as input data to other tasks. Therefore the driver ports are also called the *input drivers* of the tasks. The E Machine also communicates to the S Machine the tasks that it has released for execution via the *release ports.*

The S Machine uses the data received from the E Machine via the release ports along with the S Code to decide when to dispatch tasks. It does this through the *dispatch ports* using the features offered by the specific platform.

The **E Code** [5] and **S Code** [7] instructions would have to be understood to explain the design of the E Machine. A description of the relevant instructions is given below.

E Code has three primary instructions taking care of task execution

1. *Call driver*: Synchronous system level call to initiate the execution of any driver. A driver could indicate a task output, task input, sensor, or actuator driver.
2. *Schedule task*: Signals the S Machine via the release port that a task is ready for execution.
3. *Future time, E Code*: Marks a block of E Code for execution at a future time. It has two parameters- a trigger and the address of the E Code from where to begin execution next. Although the E Code syntax allows for a trigger to take any arbitrary form, as in a sensor input, since we are following the time-triggered paradigm, all triggers are constrained to be time triggers. The unit of time in this design is milliseconds.

In addition to these three instructions there are also *if-then-else* constructs and the unconditional *jump* instructions.

S Code again has three instructions used to decide task schedules.

1. *Dispatch task*: Initiates task execution. The task wakes up, executes the instructions required of it in that invocation, and then puts itself to sleep.
2. *Idle Time*: The *Idle* instruction divides up S Code into blocks in a similar fashion to how the *future* instruction divides up E Code into execution blocks. It indicates the time to wait for the tasks dispatched in the block above to the S Code interpreter. Each task dispatched gets an equal fraction of the time specified (in milliseconds) in the Idle instruction. If, however, non-uniform distribution of execution time is desired, it can be achieved by staggering Idle statements and Dispatch statements.
3. *Fork* and *Return Address*: These are two separate instructions that when used together constitute a *Goto* instruction. If a fork statement is used by itself to point to another block of S Code, it implies that that block execute concurrently with the block following the fork statement. This introduces non-determinism as we cannot then determine the exact time line of the tasks from then on. Therefore, usage of the fork instruction without the return instruction is prohibited.

2.1.2 Design of the ES Machine

The ES Machine primarily consists of the E & S Code interpreters. The Explicit Plan Scheduler was used to realize the interpreters.

The Explicit Plan Scheduler

This scheduler accepts a data structure containing the following fields for every task:

```
typedef struct interval {
     int processor;
     struct timespec begin;
     struct timespec end;
     int repeat;
     struct timespec period;
} interval_t;
```

The first field `processor` is aimed at SMP systems and can be ignored. The second and third fields `begin` and `end` are used to specify the beginning and end times of the task relative to the time of submission. `repeat` specifies the number of times the interval shall repeat with a period given by `period` relative to the `begin` time. All times are specified in nanoseconds.

When a task is invoked by the scheduler, it carries out the functionality required of it for that invocation and goes back to sleep. It does this by a system call which suspends the task. If the task has not finished execution by the time the `end` time is reached, it gets suspended by the explicit plan scheduler. Therefore, it is important that the time given to the task to execute exceeds the WCET of the task. A task suspended in the course of its execution by the scheduler finishes the remainder of its execution upon its next invocation before suspending itself. In doing so, it has fallen back one iteration.

As can be seen, the exact time of each and every invocation of a task must be specified with this scheduler. This creates a problem. The original S Code semantics proposed by Kirsch, et. al. [7] would wait for the completion of the dispatched task before continuing. To give an example, the S-Code

```
r:Dispatch A
  Dispatch B
```

```
Idle 6
Fork r
Return
```

would result in the two tasks A and B executing with a total time of execution of 6 ms. However, the exact invocation time of B, and the execution times of A and B are not individually known. The S-Code 'waits' till A finishes execution before invoking B. Therefore, A can execute for 1 ms leaving B 5 ms for executing. On the other hand, A could take 5 or even 6 ms leaving B 1 ms or no time at all to execute.

However, the usage of the explicit plan scheduler forces the specification of an end time for the task in advance. Incorporating the 'wait' feature into the S Code interpreter while using the Explicit Plan Scheduler would result in an awkward and inefficient design. A possible circumvention would be to design a new scheduler and plug it into the KURT kernel. While this might resolve the problem, a concern is that the strict adherence to the absolute time scale which the explicit plan scheduler brings about would be lost. Therefore, a decision was made to go ahead with the explicit plan scheduler.

The semantics of the S Code was modified to make the execution times of the tasks explicit. As stated in the explanation of the Idle instruction above, each idle instruction results in the time specified as the argument to it being divided up equally and distributed among the tasks involved. The above S Code would result in the two tasks A and B each being given 3 ms to execute. This makes the variations in task dispatch patterns carried out by S Code running on this interpreter a stricter subset of that initially proposed. However, these semantics are more deterministic, in a sense. Kirsch's S Code gives no indication as to the invocation time of B, nor does it give an idea of how much time the tasks would run for. The advance knowledge of invocation times of tasks can in certain places be desirable. A good example of such an application is the audio one discussed in the Results section, which had periodic feeding of data to an I/O device at fixed intervals.

These semantics also allow less overhead on the part of the interpreter. Previous designs of the interpreter [8] and the possible new scheduler of KURT to incorporate this feature would have the S Machine waking up after the execution of every task to schedule the next. In the present design, once the timelines for all the tasks are submitted

to the scheduler, the interpreter can sleep until all tasks have been invoked and executed, reducing overhead. However, the modified semantics requires more work on the side of the compiler generating the S Code as it will have to calculate the time line of each task.

Since the interpreters know the whole timeline of execution, including when they themselves should execute succeeding blocks of E Code or S Code, an optimal design would have been a self-scheduling ES Machine, where the interpreters schedule the tasks and suspend themselves setting their next invocation when the next block of E Code or S Code should be read. However, such a design has a problem.

As mentioned above, all times specified in the explicit plan structure are relative to the submission time. If the interpreter submits it, the time of submission is an uncertain quantity as the time taken by the ES Machine to reach the submission routine would depend on the number of instructions it has read through and interpreted until then, which depends on the code being read. Therefore, figuring out a method of dispatching tasks with respect to an absolute time line would have been hard. To surmount this, the present design was developed:

The Heartbeat Thread

At the heart of the ES Machine lies a timer thread, a heartbeat, which is set up during initialization with a recurring execution and a fixed interval. The heartbeat thread serves as a reference thread. Since it is set up with a fixed interval in the beginning, the explicit plan scheduler invokes it with precision each time. Thus the exact time of submission is a known quantity as the only instruction this heartbeat executes is the submission of the schedule created by the interpreters.

The Thread Structure

The primary heartbeat thread is set up during initialization of the E Machine to repeat with a fixed interval. The interval is given by the `SMACHINE_EXEC_INTERVAL` macro which is part of the S Code program and is the highest common factor of all the different intervals of execution of the S Code. To give an example, if in an S Code program there are idle instructions having times of 16 and 20 milliseconds, then `SMACHINE_EXEC_INTERVAL` can have a maximum value of 4. The rationale behind having the heartbeat thread dependent on the S Code is as follows. The E Machine

releases tasks to the S Machine. The S Machine divides up the interval specified by the E Machine, within which all the released tasks must execute, into finer time slices and assigns them to tasks. Therefore the S Machine must necessarily have a higher frequency of invocation than the E Machine.

Now the heartbeat thread must be invoked after every S Machine execution to submit the schedule generated. Therefore the largest possible value of `SMACHINE_EXEC_INTERVAL` is the highest common factor of all the possible intervals of the S Machine. Since each invocation of the heartbeat causes a certain overhead on the system, it is in the interests of performance to have the heartbeat as low as possible, and consequently, for `SMACHINE_EXEC_INTERVAL` to be as high as possible. Since the recurring schedule of the heartbeat thread has to setup during the initialization phase, it forces a static value of `SMACHINE_EXEC_INTERVAL`. This is ok as the E/S Code is being compiled in with the ES Machine, forbidding run-time modifications to the E/S Code invocation times.

The second system thread, called the ES Machine thread, has the interpreters for both E Code and S Code. This is invoked for every S Machine execution. By design, the E Machine execution time should coincide with one of the S Machine execution times. During an invocation of this thread, *Algorithm 1* is executed.

```
if E Machine execution time has passed then
     throw illegal time line exception
else if at least one E Machine trigger is active then
     call E Code Interpreter
end if
call S Code Interpreter
suspend self
```

Algorithm 1: The ES Machine Thread

The above algorithm introduces the concept of *triggers*. A trigger is what activates a particular block of E Code and starts its interpretation. Since we are considering a time-triggered architecture, all triggers are time triggers. A trigger is a tuple

consisting of the address of an E Code location, and a positive integer denoting the time the E Code at the address becomes active. The time is calculated relative to the time of submission. The E Machine maintains a queue of such triggers in chronological order, and the first trigger in the queue is checked by the ES Machine thread in it decision to wake up the E Code interpreter.

E Machine – The E Code Interpreter

The algorithm for the E Code interpreter is described already in [5]. A simplified version is given in *Algorithm 2* for the sake of being self-contained.

```
while ProgramCounter != END_OF_PROGRAM
     i = instruction (ProgramCounter)
     ProgramCounter = next (ProgramCounter)
     if i = call(d) then
          if driver d accesses a port of a task that has
              been  released but not completed then
               throw time safety execution
          else execute d
     else if i = schedule(t) then
          if task t has been released but not completed
          then throw time safety execution
          else release t to the S Machine
     else if i = future(pc,time) then
          append (pc,time) to trigger queue
     end if
end while
```

Algorithm 2: The E Code Interpreter

S Machine – The S Code Interpreter

The algorithm for the S Code interpreter is given by *Algorithm 3*.

```
while ProgramCounter != END_OF_PROGRAM
i = instruction (ProgramCounter)
ProgramCounter = next (ProgramCounter)
if i = dispatch(t) then
     if t ∈ task set released by E Machine then
          remove t from task set released by E Machine
          add t to task dispatch set
     else if t is not finished executing
          add t to task dispatch list
else if i = idle(time) then
     n = no. of tasks in task dispatch set
     time per task = (time – ES Machine time) / n
     time line = |
     counter = 0
     while (task dispatch set != ∅)
          remove the first task t from task dispatch list
          time line = time line U {t, counter : counter+n }
          counter = counter + n
     time line = time line U ES Machine time
     set up time line to be submitted to the Explicit Plan
     scheduler upon the next invocation of the timer thread
else if i = fork (pc) then
     if next(i) = return then
          set ProgramCounter = pc
     else
          throw illegal opcode exception
else
     throw illegal opcode exception
end if
end while
```

Algorithm 3: The S Code Interpreter

In this algorithm the time 'ES Machine time' is a macro in the code which is platform specific and has to be set by the user of the ES Machine. It indicates the time taken by the ES Machine to execute. Presently, a generous worst case execution time of 600 microseconds has been set. This is the time given for the execution of both the interpreters as well as the timer thread. The time actually taken by these tasks when we instrumented them was not more than 60 microseconds. This number can change depending on the data types the drivers, which execute in a synchronous fashion, deal with. As shall be explained later, drivers have to copy data between arrays which has an impact upon the interpreters' execution time. The figure of 60 microseconds was obtained with simple integer data types. Therefore, a safety factor of 10 was built in.

In the algorithm, the line

```
time line = time line U ES Machine time
```

hides behind it a few implementation specific details such as setting up the ES Machine for execution while giving the recurring timer thread enough space such that the time line of the ES Machine does not intrude into that of the timer thread. All these parameters such as the time required by the interpreters, timer threads etc are platform specific and are declared as macros in the file `os_interface.h` in the `c_platform/emachine` subdirectory under the root ES Machine directory. A look at the file reveals the macros:

```
#define TEST                        (0)
#define SCALE_DOWN                  (1)
#define ES MACHINE_EXEC_TIME        (400 * SCALE_DOWN)
#define ADMIN_THREADS_EXEC_TIME     (600 * SCALE_DOWN)
#define INTERVAL_START_DELAY        (50)
#define TIMER_EXEC_TIME             (200 * SCALE_DOWN)
```

`TEST` is used for testing purposes without making use of the KURT API. In that mode it displays the interval times calculated and waits for user inputs to execute each block of E Code and release tasks.

`SCALE_DOWN` is to slow down the ES Machine, again for testing purposes

`ES MACHINE_EXEC_TIME` is the time given to the E&S Code interpreters to execute

`TIMER_EXEC_TIME` is the time given to the timer thread to run

ADMIN_THREADS_EXEC_TIME is the sum of the above two times

INTERVAL_START_DELAY is the offset given to all the task and ES Machine time lines when they are calculated. The schedule for the tasks and the ES Machine thread is submitted sometime during TIMER_EXEC_TIME. However, the submitted invocation times should occur only once TIMER_EXEC_TIME has expired. This offset makes sure of that.

All the times are specified in microseconds.

2.1.3 Step-Through - An example Iteration

A typical timeline of ES Machine execution is outlined below.

Initialization

When the ES Machine is in the initialization phase, it creates the E Machine group required by the group scheduler, affiliates it to the Explicit Plan scheduling decision function; initializes task semaphores required for maintenance and information gathering; creates the interpreter thread and the various task threads.

The main ES Machine thread then becomes the timer thread and enters the heartbeat loop. It does this by setting itself up for periodic invocation every SMACHINE_EXEC_INTERVAL microseconds, each invocation being of TIMER_EXEC_TIME microseconds duration. It also sets up the ES Machine interpreters' thread for it's the execution of the first blocks of E Code and S Code.

The ES Machine Thread

The ES Machine thread first calls the *E Machine* or *E Code interpreter*. The E Code interpreter peruses the E Code calling the initialization and input and output drivers of the various tasks synchronously. Each time a schedule task command is reached, it adds it to the list of released tasks to hand over to the S Machine. This continues until the future command is encountered. Then the next invocation time of the E Machine is stored and control is returned to the ES Machine thread.

The *S Machine* or *S Code interpreter* is then called. It reads the S Code following the algorithm outlined in Algorithm 1, creating the time line structure from the time of the interval submission (i.e. the next invocation of the timer thread) to the next invocation of the ES Machine thread. The time line structure, in addition to containing the

invocation times of the dispatched tasks, must also contain the next invocation of the ES Machine thread so that the next block of E or S Code can be processed.

The Timer Thread

The timer thread then gets invoked and submits the time line created by the S Machine to the explicit plan scheduler, following which it suspends itself until invoked again due to its recurring schedule.

The Task Threads

Once the interval is submitted, the tasks which had been dispatched by the S Machine are invoked and carry out one iteration of their routine each before suspending themselves.

This cycle repeats with the ES Machine thread getting invoked and creating the time line for the next invocation of the task threads and itself.

An Example

Consider two tasks A and B with A generating data and feeding it to B. Let's suppose B has a frequency twice that of A. Assume a Giotto period of 10ms and A having a frequency of 1 and B, 2. i.e. A has a period of 10ms, while B has 5.

The E Code for such a system will be:-

```
p:Call output driver A          q:Call output driver B
  Call output driver B            Call input driver B
  Call input driver A             Schedule B
  Call input driver B             Future 5, p
  Schedule A                      Return
  Schedule B
  Future 5, q
  Return
```

Various S Codes are possible. The simplest one would be:

```
r:Dispatch A
  Dispatch B
```

```
Idle 5
Fork r
Return
```

This code would result in the following time line:

No.	Time(in μs)	Task
1	t	ES Machine execution
2	+400	Timer thread interval submission
3	+200	Task A
4	+2200	Task B
5	+2200	ES Machine Execution
6	+400	Timer thread interval submission
7	+200	Task B
8	+4400	ES Machine Execution

The 2 cycles (1-4 & 5-7) shall repeat if task A has completed execution during the first cycle itself. However, if during the invocation of the ES Machine in the second cycle, it sees that A has not completed, it is scheduled again, resulting in the first cycle (1-4) repeating itself. This is in keeping with the semantics of Giotto, as A has a time period of 10ms, and therefore has 10ms to finish one invocation. This however results in the invocation times of B being staggered if A finishes execution in time. The intervals between B's invocation times can vary between 2800μs (4-7) and 7200μs (7-8, 1-4). Though the task finishes within the deadline (assuming, of course, a WCET<2200μs), it might be desirable to have tasks begin on a periodic basis too. If such is the case, then an alternate block of S Code such as the one below can be used :-

```
r:Dispatch A
   Idle 2.5
   Dispatch B
   Idle 2.5
   Fork r
   Return
```

This code would result in the following time line

No.	Time(in μs)	Task
1	t	ES Machine execution
2	+400	Timer thread interval submission
3	+200	Task A
4	+1900	ES Machine Execution
5	+400	Timer thread interval submission
6	+200	Task B
7	+1900	ES Machine Execution
8	+400	Timer thread interval submission
9	+200	<idle>
10	+1900	ES Machine Execution
11	+400	Timer thread interval submission
12	+200	Task B
13	+1900	ES Machine Execution

Here we obtain periodic execution times for task B with constant intervals of 5000μs between invocations. The price paid is that if A needs only 1900μs or less to complete its execution, then one slot of 1900 will lie empty every alternate cycle. Also both tasks get a reduced execution time (from 2200μs to 1900μs).

2.1.4 Conclusion

The ES Machine when run on the KURT-Linux platform displays timing properties stated in the original model. However the ES Machine can only execute the tasks in real-time if the timing requirements stated in the original Ptolemy II model allow the tasks to finish execution. To verify the time safety of the model, the WCET of the tasks have to be provided to the Giotto compiler as annotations in the Giotto code. This will have to be calculated on a per-platform basis and supplied separately by the designer.

The ES Machine requires as input to it

- The task code in C
- E Code

- S Code
- Framework code consisting of driver code for all the ports

In the next section which details the first stage of implementation, it shall be shown how these are generated.

2.2. The ES Machine Code Generator

2.2.1 Introduction

The ES Machine code generator generates code for the ES Machine starting from a Ptolemy II Giotto model. A sample Ptolemy II Giotto model is shown in figure 3.

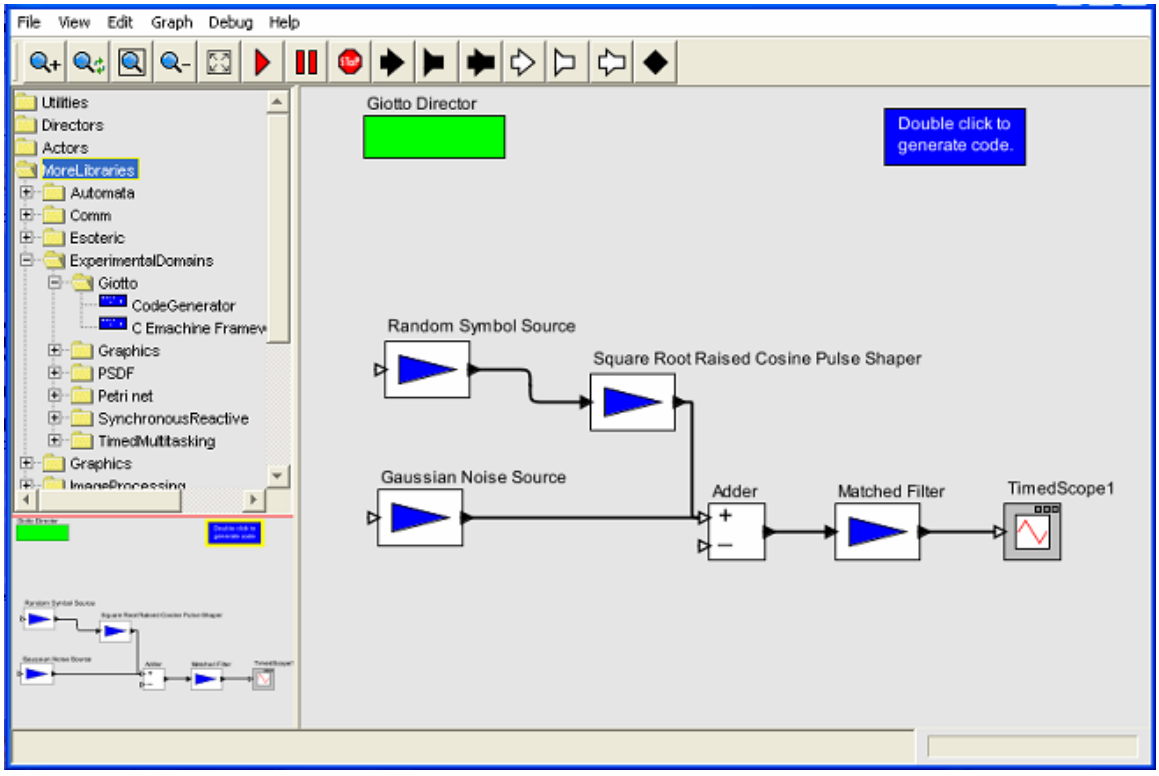

Figure 3: Sample Ptolemy II Giotto model

As mentioned in the introduction, Ptolemy *actors* represent Giotto *tasks*. The *ports* of the actors are analogous to the ES Machine ports, with the output ports representing the task ports, and input ports the driver ports. An input port into the model

is used to represent a sensor and an output port, an actuator. The *relations* show the communication of data between ports, from sensors and to actuators.

The Ptolemy II model can simulate the behavior the final executable will display. The Giotto director takes care of scheduling the actors (tasks) as per the Giotto semantics.

2.2.2 Design Considerations

The ES Machine requires 4 pieces of code from the Code Generator to function.

The *Framework Code* requires analysis of the structure of the model to generate the appropriate driver code to transfer data between tasks, from the sensors and to the actuators.

The *E Code* can be generated by the Giotto compiler developed in the Giotto project, by compiling a *Giotto Program*. Therefore, it is enough to generate a Giotto program from the model.

The *S Code* requires a scheduling algorithm to come up with a feasible schedule for the set of tasks. However, this is not the focus of this project. There already exists a good body on work on the topic of scheduling. For a simple schedule an extension of the first S Code given in the example in section 2.1.3 can be applied. For now, it shall be assumed that the designer writes the S Code.

The *Task Code* is a problem of compiling a Ptolemy II model down to C code. There is an ongoing effort in the Ptolemy project to generate code from graphical models using the Copernicus tool developed by Stephen Neuendorffer. It is focused on the generation of java code. Extending this tool to generate C-code could be one possible solution to generating task code. However, another design methodology has been adopted that shall be described in the next section.

2.2.3 Design

CActors and CPorts

A new actor termed *CActor* is introduced to be used in Giotto models. This actor is a Ptolemy II wrapper for functions written in C. Software developers are required to link each of their tasks written in C with an instance of CActor. This actor is then added to a library of such actors.

In order for the actors and the ports to lend themselves to having framework and Giotto code be generated, a few specific parameters are required. They are listed as follows.

Model Parameters

period: The time period of one iteration of the Giotto model. This Giotto super-period is specified in the director. It defaults to 1.0 second.

frequency: The number of times a task is executed in the time period *period*. This is specified in the actor representing the task. It has a default value of 1.

_type: The type of the output port. Presently data types of int and double and their arrays are supported. Input ports do not have to specify a type as it is taken to be the same as that of the output ports they connect to.

The above three parameters are required by the Giotto code generator. The following two parameters are required by the Framework code generator to generate C code and are ES Machine specific.

initialOutputValue: The initial value which an output port shall have before the first execution of the task it belongs to assigns it one. This is necessary in case a task having this port as one of its inputs executes before the task which assigns the port a value. It defaults to 0.

arrayLength: The number of elements in the array if the port type happens to be one. If the port type is not an array this parameter is ignored. It defaults to 1.

Since the designed application is intended to be a real time system, dynamic memory allocation should not be used. Static allocation of the memory before the model runs requires *a-priori* knowledge of the maximum number of elements the array might have at any point during the execution of the model. This is a figure only the developers of the tasks would know and so this parameter was introduced to inform the framework code of the array sizes. The framework code then takes care of statically allocating the requisite memory spaces. All tasks are passed pointers to the allocated regions. This prevents the user having to worry about allocating memory and makes it easier for the ES Machine to manage data.

Data Transfer Control

As per Giotto semantics the data output by a task is visible to other tasks and the outside world only at the end of the task time period. Since a task can only read visible input data, only those tasks which get invoked after the time period of a task execution ends can read the data from that invocation. This is implemented by maintaining multiple copies of each data element.

Copy1, or the output copy, is visible only to the output port of the task that provides the data.

Copy2, or the global copy is visible to the entire system.

Each input port that connects to that output port has a copy which is visible only to the port.

Transferring data in a timely manner between these copies is the task of the ES Machine.

Whenever a task is invoked, it is given a pointer to all its output and input ports. The E Code calls the output driver of a port when the time period of the task containing that port has expired and the input driver of a port when the corresponding task's time period is about to begin.

The framework code maintains a memory area for each copy of the data. The output copy is initially made to point to memory 1, the global copy, memory 2, and the input copies made to point to their separate memory areas. When the output driver is called, the pointers of the output and global copy are simply swapped, thus making the output data visible, and giving the task a fresh slate to write the data on for the next time period.

Whenever the input driver is called, however, a memory copy from the global copy to the local memory pointed to by the input port is performed. A pointer swap here would prevent multiple tasks from acquiring the same data should they all be connected to the same output port.

The same mechanism is applicable to both single data elements like double, as well as array data types. Synchronization is not a concern as only the interpreter calls the various driver functions.

Giotto Code Generation

There already exists in Ptolemy II a Giotto code generator which would take in a Giotto model and display Giotto code in a text window. This was sufficient for the most part. Therefore it was sublcassed and modified.

The derived class is called *GiottoCEmachineFrameworkGenerator* and supports arrays and integer data types in addition to the double data type supported by the base class. While extending support to integers was trivial, support for arrays was accomplished by defining a new data type for each array type in the Ptolemy II model.

An array type is declared in Ptolemy II by enclosing the data type of the array elements within '{' and '}'. This was transposed to a data type whose name consisted of the elements' data type followed by the string 'array'. For example, an array of integers is declared in Ptolemy II with the string {int}. In the generated code, this will result in a data type intarray being declared. This data type is declared in a header file in the framework code.

The second modification was altering the original functionality of displaying the Giotto code in a text window, to asking the user for a directory to place files in and writing the code to a file.

Framework Code Generation

The framework code generator produces 3 files.

f_code.c: This file contains all the driver code. This includes the initialization code which assigns the initial values for the output ports, code to allocate memory for all ports and input and output driver code for data transfer.

f_code.h: This file contains the declarations of all the functions defined in f_code.c, and all data type declarations.

task_code.h: This file contains the declarations for the task functions. This is mainly done so that the developer who writes the C-code for the functions can know the exact signature of the functions.

2.3. Putting it together: An Example Run-Through

Consider the sample model shown in figure 3.

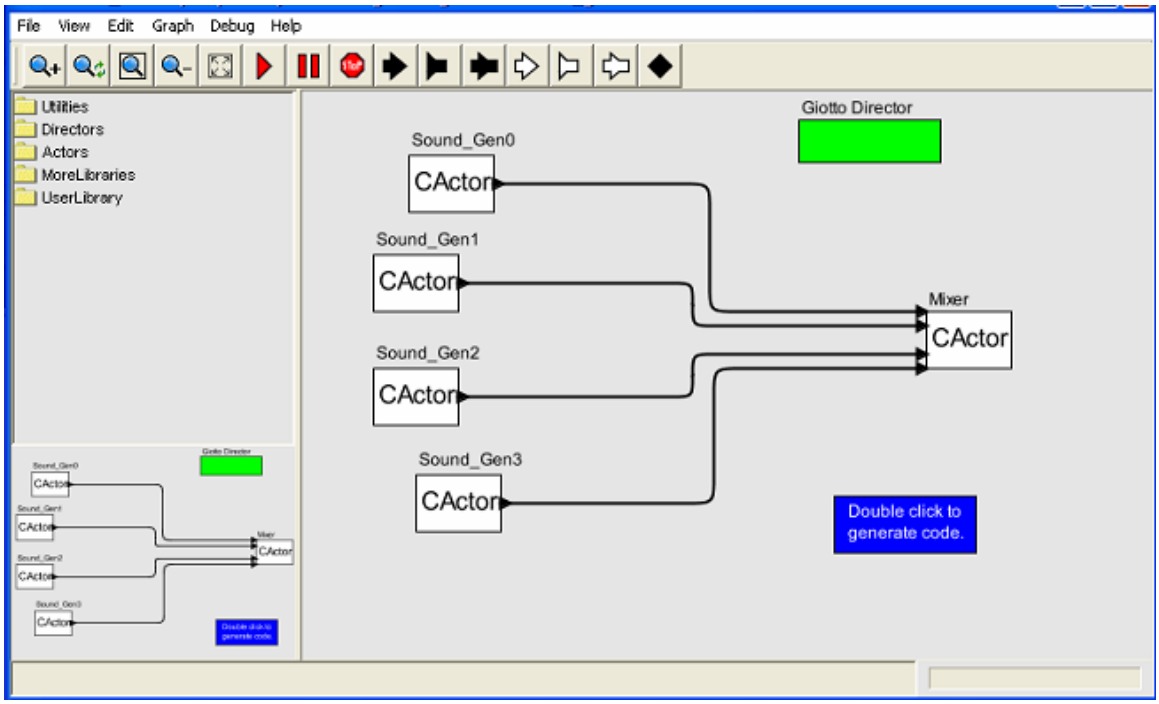

Figure 3: Sample Ptolemy II model using CActors

In the figure, we have 4 tasks (Sound_Gen0 – Sound_Gen3) generating data and feeding it into the 5th task (Mixer). All the 5 tasks are instances of the CActor class. This actor class uses instances of the CPort class as its input and output ports. A look inside the configuration dialog of the output ports of any one of the generator tasks reveals, in figure 4,

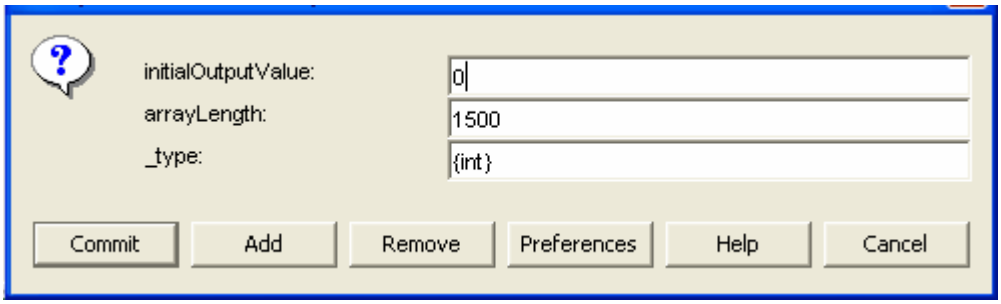

Figure 4: Configure Dialog of an output port

a data type of {int}, which is an array of type integer, of length 1500. All the ports are of this data type.

The model has a super-period of 3.2 seconds, and the generator tasks are given varying frequencies of 2, 4, 5 and 10. The mixer task, however, is given a high frequency of 100. The reason for this is given in the next section, results, where we explore an interesting interfacing aspect of this application.

Once the blue button labeled 'Double Click to Generate Code' is pressed, a dialog box opens up requiring the user to select a directory. The user selects it and then clicks the *Generate Files* button. That results in the Framework Code Generator creating the required files. In the directory chosen by the user, the code generator shall create a directory of the same name as the model. Within it, it creates the Giotto file, again with the model name. A directory structure of *c_functionality/fcode* is also created within the same directory and the three files *f_code.c*, *f_code.h* and *task_code.h* are placed in it. This directory structure has been chosen so as to mirror that of the ES Machine.

The user is then expected to feed the Giotto file to the Giotto compiler which shall compile it down to E Code and the function table mapping the E Code function calls to their corresponding driver functions. These are generated and placed in the *ecode* and *ftable* directories respectively within the *c_output* directory.

The tasks divide up the super period of 3.2 seconds into periods of 1.6 seconds (frequency 2), 0.8 seconds (frequency 4), 0.64 seconds (frequency 5), 0.32 (frequency 10) and 32ms (frequency 100). Since 32ms also divides the remaining time periods evenly, E Code blocks every 32ms to schedule tasks are sufficient to represent this design. Two block from the generated E Code, representative of the whole, are reproduced below. The whole E Code is not reproduced for the sake of brevity.

```
/* 27 */  IF(0,28,30)         /* If task driver: condition_Mixer_Mixer_driver */,
/* 28 */  CALL(8)           /* Call task driver: driver_Mixer_Mixer_driver */,
/* 29 */  SCHEDULE(4,0,131072)       /* Schedule task: task_Mixer, release
time: 0, relative deadline: 32 */,
/* 30 */  FUTURE(0,32,32)       /* Triggered jump to mode: sound_gen, unit: 3 */,
/* 31 */  RETURN()          /* From mode: sound_gen, unit: 2 */,

/* 32 */  CALL(7)           /* Call output port copy driver:
driver_Sound_Gen3_output1_copy_intarray for task: task_Sound_Gen3 */,
/* 33 */  SCHEDULE(3,0,131072)       /* Schedule task: task_Sound_Gen3, release
time: 0, relative deadline: 32 */,

/* 34 */  IF(0,35,37)         /* If task driver: condition_Mixer_Mixer_driver */,
/* 35 */  CALL(8)           /* Call task driver: driver_Mixer_Mixer_driver */,
/* 36 */  SCHEDULE(4,0,131072)       /* Schedule task: task_Mixer,
release time: 0, relative deadline: 32 */,
```

```
/* 37 */  FUTURE(0,39,32)       /* Triggered jump to mode: sound_gen, unit: 4 */,
/* 38 */  RETURN()          /* From mode: sound_gen, unit: 3 */,
```

As can be seen, the FUTURE statements (lines 30, 37) both have times of 32ms (third argument) for scheduling the next block of E Code. This pattern can be seen repeated throughout the E Code.

The first block of E Code shows a scheduling of the Mixer task, while the second task shows both the Sound_Gen3 task and Mixer task being scheduled. Most of the code is similar to block 1, a result of the high frequency of the Mixer task.

Lines 27 and 34 are conditional checks inserted by the E Code generator that call a function implemented by the designer to determine whether the task should be scheduled at all. In the present tool flow, however, this feature of the E Code is not required, resulting in the function always returning true.

Lines 28 and 35 call the input driver for the mixer task.

Lines 29 and 36 show the scheduling of the mixer task, with a deadline of 32 ms (third argument) relative to the start time.

Lines 30 and 37 direct the interpreter to schedule the next blocks (32, 39) of E Code after 32ms.

Lines 31 and 38 indicate the end of the E Code execution block, resulting in the interpreter waiting until the next scheduled block of E Code become active.

Lines 32 and 33 show, similarly, the scheduling of the Sound_Gen3 task. This does not have the input driver being called as it does not have input ports. Line 32 is a call to the output driver which copies the results of the previous execution of this task to a globally visible area. This is to comply with the Giotto semantic that the results of a task execution become available only at the end of its time period.

As can be seen, the second block schedules both the Mixer task as well as the Sound_Gen3 task. However, while the mixer task must finish execution in this time period, the Sound_Gen3 task has until its next scheduling to finish its execution. The allocation of time to these tasks is the responsibility of the S Code. A simple S Code to the effect of

```
r: Dispatch Sound_Gen0
Dispatch Sound_Gen1
```

```
Dispatch Sound_Gen2
Dispatch Sound_Gen3
Dispatch Mixer
Idle 32
Fork(r)
Return
```

will result in the Mixer task getting 32ms to execute in the first E Code block specified above, but only 16ms in the second block, the remaining 16 being allocated to the Sound_Gen3 task, as shown in figure 5.

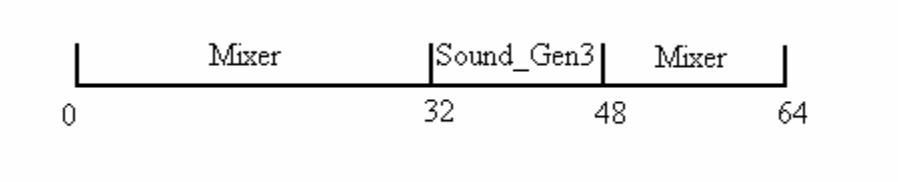

Figure 5: Timeline of the two E Code blocks

The mixer task required less than 2ms in this application, and so this S Code would have been sufficient resource allocation-wise. Similarly at every 1600 milliseconds, all the 5 tasks will be scheduled at once. This will result, with the present S Code, in each task getting only 6ms to execute. This, again, is sufficient for the Mixer task. The other tasks have opportunity to execute in following 32ms time spans, and therefore completing in time.

However, the Mixer task was interfacing with external devices and it was desirable that its invocation times were periodic. Towards that purpose, a modified S Code of

```
r: Dispatch Sound_Gen0
Dispatch Sound_Gen1
Dispatch Sound_Gen2
Dispatch Sound_Gen3
Idle 28
Dispatch Mixer
Idle 4
Fork(r)
Return
```

was used.

Here, if none of the Sound_Gen tasks needed to run, then the first period of 28ms would have the system being idle. This is not a waste, as the Mixer task will anyway use only 2ms. However, this S Code has the benefits of the exact invocation times of the Mixer task being known, it being regular, and the Mixer task getting 4ms regardless of the number of other tasks that need to run. Further fine graining the S Code is definitely possible, but was not required for this application. The timeline now obtained is shown in figure 6. SG0-3 represent tasks Sound_Gen0-3 and M represents the Mixer task.

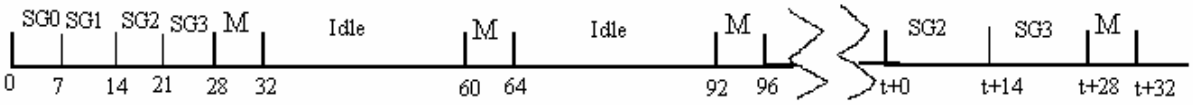

Figure 6: Timeline of the tasks in the example application

The last stage in the run through is to copy the generated files and place them within the ES Machine directory structure. Since the generated files have all followed the directory hierarchy all that needs to be done is to copy the c_output and c_functionality directory. The S Code file is located in *c_output/scode/smachine_code.c* and will have to be modified by hand presently. There exists a default S Code which simply assigns each task an equal time slice. If further fine tuning is required, the file can be edited. The E Code interpreter can be found in *c_platform/emachine/e_machine.c*, and the S Code interpreter in *c_functionality/scode/s_interface.c*. The platform specific parameters to tweak the timing of the E&S Machines can be found in *c_platform/emachine/os_interface.h*.

The task code also needs to be copied, preferably into *c_functionality/fcode.* Their corresponding header files should include *task_code.h*.

Once the files are copied, they can be compiled on KURT-Linux using KURT libraries (-lkurt option in gcc) to provide the executable.

3. Results

The ES Machine was instrumented using the DSUI feature provided with KURT. Measurements showed that all the threads, including the task, ES Machine and timer threads were being invoked at the expected times with a jitter of ±30μs.

A Discussion on interfacing the ES Machine with I/O

An example application was developed and run to measure the effectiveness of the tool chain. The Ptolemy II model is the same one shown in figure 3. The Sound_Gen tasks generate sound data in PCM format corresponding to the plucking of a string. They utilize the Karplus-Strong algorithm [10] to do so. The Karplus Strong periods are different for each task thereby producing sounds of different pitches. The data from these four tasks are fed into the mixer. The purpose of the mixer is to merge the audio data to produce one final PCM waveform which has sounds from all the generators. The mixing is weighted to give the sound from generator 3 a slightly higher volume. This will help the listener hear the beat from generator 3 more clearly and can be used to verify the timing of the task. The mixer task interacts with the sound card using the ALSA driver [11] which provides a low level API with low latencies.

The model had a super-period of 3.2 seconds, and the generator tasks were given varying frequencies like 2,4,5 and 10 to try out different beats. The mixer task, however, was given a high frequency of 100. This was primarily to test the capability of the application in keeping up with timing requirements. The mixer task would operate with a period of 32ms. Correspondingly, the ALSA API was set up to receive sound samples every 32ms.

Setting up ALSA required setting up the buffer and periods to receive data in frames. A *frame* consists of the data equivalent of one PCM sample per channel. For a stereo system (2 channels) receiving PCM data sampled at 16 bits, the frame size is 4 bytes. Sound data in frames is given to the ALSA driver through the buffer. The *buffer* is the memory area allocated by the ALSA driver to read data from. A buffer is divided into *periods*. Data is read and written on a per-period basis. The driver locks a period while reading from it preventing applications from writing to it. Therefore, ALSA has a minimum requirement of two periods such that an application can write to one while ALSA reads from the other.

Once the driver is initialized, it lies idle until the first period is written into. Once it receives data, the driver starts to read it and feed the soundcard continuously until an API call is made to stop. Once the reading has begun, the ALSA driver must necessarily receive enough data to have a non-empty buffer. If the buffer ever becomes empty, the driver responds with a buffer underrun error message.

The application ALSA setup consisted of the following:

- A PCM Rate of 16000Hz
- A buffer of 1024 frames consisting of two periods of 512 frames each.

This configuration requires a period to be filled every 32ms. The mixer task that writes into the buffer does so every 32ms with 512 frames of data; in other words, just enough data to prevent the driver from starving. If the data was late by more than the time required by the ALSA driver to read one sample, which in the case of this configuration is 62.5μs, the driver would report a buffer underrun.

Running this application resulted in errors every time. ALSA kept reporting buffer underruns. DSUI instrumentation of the application showed timely executions. The jitter factor of 30μs was within the forgiveness time of the ALSA driver (62.5μs). After repeated testing, the only feasible reason remaining was clock drift, i.e. the CPU and soundcard clocks are different which could result in underruns if the soundcard clock ran at a slightly higher rate than the system clock. One of the proposed solutions to counter clock drift was as follows:

The application was redesigned with the mixer task running with a period slightly less than that required by the ALSA driver. The ALSA driver was also setup to run in the blocking mode. This means that if there is not enough free space available, a call to write into the buffer blocks until space is freed up by the ALSA driver as it reads out data.

An upper bound of the rate difference between the two clocks of 1 in 32 was assumed. The created Giotto model therefore had the frequency of the mixer task set to 31ms while writing 32ms worth of sound data each time to the ALSA buffer. This was accomplished by having a Giotto period of 3100ms and a mixer task frequency of 100.

An underrun is obvious with a rate difference greater than 1 in 32 in favor of the soundcard clock. However, what happens if the rate difference is less than 1 in 32? The

paragraphs below make an argument that the above model is enough to prevent underruns in such a case.

The criterion for an underrun is to have an empty buffer for longer than 62.5μs. In the scenario where the model has just begun executing, an underrun cannot happen for two invocations of the mixer task since they will occur before the ALSA driver can finish reading the written data. The buffer has then filled up blocking the mixer task when it makes the third write call. That will result in the mixer task being suspended until the next iteration. Upon being dispatched again by the S Machine, the mixer task simply finishes the writing process and suspends itself. It has now missed one time period. This is the only possible case when an underrun can occur. A closer look at this case reveals the following:

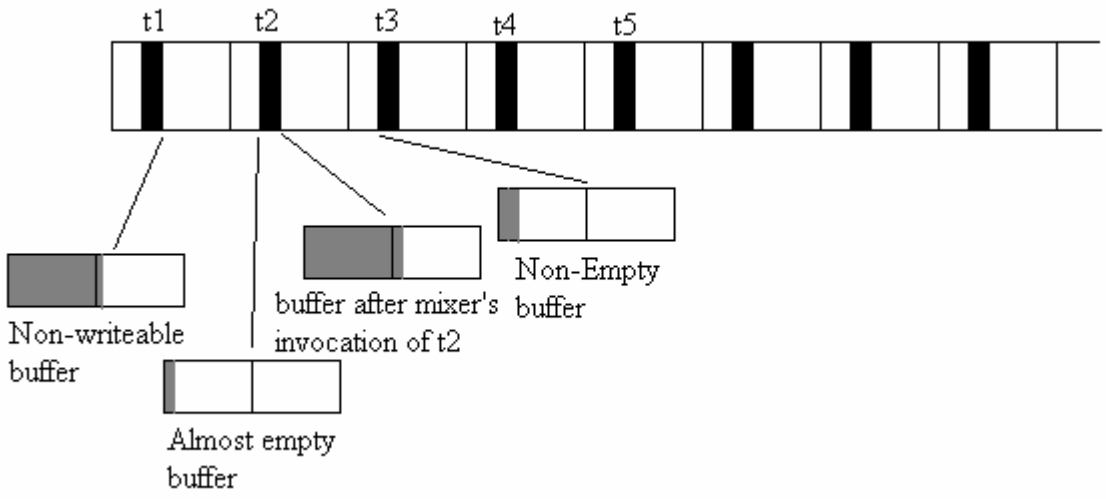

Figure 7: Timeline of the Mixer task in the ALSA example

In figure 7, each rectangle represents one hundredth of a Giotto period with the solid black bars inside representing the execution of the mixer task. t1,t2,t3, etc are the begin times of the mixer task invocation and are therefore separated by 31ms each. The worst case scenario we can imagine at t1 is the ALSA buffer having one period completely full and the other one with one frame of data, which is being read by the driver. This will prevent data being written to the buffer and the mixer task call to write shall block.

After 31ms (minus the execution time of the mixer task), at t2, the buffer will have one period empty and the other one with more than one sample remaining, as a result of the boundary condition of the drift being less than 1ms. Even assuming a mixer task execution time of zero, and a clock rate difference only infinitesimally smaller than 1 in 32, there must be at least one sample in the buffer. i.e. there cannot be an empty buffer at t2. Once the mixer task has finished this second execution, there again will remain a buffer that has one period full with the other period having one sample. At t3 and further time periods, the situation will still be that which existed at t2.

Thus the only way for an underrun to happen is if the timing inaccuracy measured using DSUI of a maximum of 30 μs results in two invocations being separated by an extra 60μs which means that the rate difference becomes more than 1 in 32. In other words, if the frequency of execution of the mixer task is such that it feeds in data to the ALSA buffer at a rate greater than the read rate of the driver, then we shall have no underruns.

The application thus created was executed and no underruns were observed. However, it is only fair to mention that readings were only taken for an hour. This approach to having 2 asynchronous systems communicate with each other without the use of any explicit feedback mechanism shows how skipping a deadline can in some cases aid in the proper functioning of a real-time system.

4. Limitations

The software in its present incarnation has a few limitations

- The present design requires the usage of the heartbeat or timer thread. This was necessary because when submitting Explicit Plan schedules, a reference time is required. All times are relative to the time of submission requiring the need for a reliable ‘tick’ to calibrate the submission. This can be seen as a suboptimality in the design of the ES Machine as there can be unnecessary invocations of the timer thread taking CPU time away from the tasks.

- Instrumentation using DSUI is not possible in some cases as certain function calls made when the instrumentation mechanisms are active cause the system to hang. An example is the *strdup* call or the *snd_pcm_open* call when ALSA is compiled in along with DSUI. The reason for this was not clear at the moment this report was written.

5. Summary

This project report describes an extension to the Ptolemy II modeling and simulation software. The added functionality generates an executable from a Ptolemy II Giotto model that when run on the KURT-Linux platform exhibits timing characteristics faithful to the requirements of the original model. Applications produced using this software showed an error in the timing characteristics of less than ±30 microseconds.

Also described is an example application which interfaced the ES Machine with the ALSA audio driver to produce rhythmic string-plucking sounds. A problem faced in spite of correct software design due to the peculiarity of the hardware underneath was stated, and a possible software solution to it was described. This approach showed how missing of deadlines by software tasks might in some cases be necessary for the proper functioning of the system as a whole.

6. Future Work

Extensions to the Giotto model in Ptolemy II involve the development of a suitable scheduling algorithm can also be imported and modified to automate the writing of S Code.

A longer term extension to this work is the inclusion of code generation mechanisms to generate task code in C. Once that is incorporated, the Ptolemy II model simulation can be made to occur in two ways.

Presently various Ptolemy II actors from the existing library are used to create a model. The present simulation occurs with these actors that can be compiled down to tasks in C code and then associated with CActor instances. By placing these CActors in the Giotto model, the Giotto director can possibly be extended to simulate this model wherein the execution of C-code shall take place through a JNI interface.

Other extensions include adding support for sensors, actuators and multiple modes. Multiple modes are presently not supported by the ES Machine code generator. The ES Machine, however, supports it. The reason why it was not included in this version is that the notion of modes in Ptolemy II and Giotto are different. Ptolemy II purports a hierarchical notion of modes, while Giotto does not have a notion of hierarchy. The incorporation of modes into a Ptolemy II model as Giotto specifies it requires a model

utilizing the FSM model of computation with each of its states being a Giotto model. These states then represent the different modes in which the Giotto model can be. This however represents only a subset of the Giotto specification as the changing of modes in the middle of a Giotto iteration is not supported. It can be argued that such a constraint actually frees the designer from creating a potentially error prone design. According to Giotto, if a mode changes in the middle of the execution period of a task, then the new mode must contain the same task with the same execution period. Constraining mode changes to occur only in between periods removes this constraint. Two possible approaches to include modes are either redesigning the ES Machine code generator to recognize the above mentioned hierarchy and translate it to how Giotto understands it, or to redesign Giotto itself to include a hierarchical notion of modes.

7. Bibliography

[1] J. Davis, C. Hylands, J. Janneck, E.A. Lee, J. Liu, X. Liu, S. Neuendorffer,S. Sachs, M. Stewart, K. Vissers, P. Whitaker, Y. Xiong. "*Overview of the Ptolemy Project*". Technical Memorandum UCB/ERL M01/11, EECS, University of California, Berkeley, March 6, 2001.

[2] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. "*Giotto: A time-triggered language for embedded programming*". Proceedings of the IEEE, 91(1):84–99,2003.

[3] R. Hill, B. Srinivasan, S. Pather and D. Niehaus. "*Temporal Resolution and Real-Time Extensions to Linux*", Techreport, kurt-utime-1998, ITTC-FY98-TR-11510-03, "Information and Telecommunication Technology Center, University of Kansas", June, 1998

[4] W. Elmenreich and G. Bauer and H. Kopetz. "*The Time-Triggered Paradigm*", Proceedings of the Workshop on Time-Triggered and Real-Time Communication, Manno, Switzerland, Dec, 2003.

[5] Thomas A. Henzinger and Christoph M. Kirsch. "*The Embedded Machine: Predictable, portable real-time code*". Proceedings of the International Conference on Programming Language Design and Implementation (PLDI), ACM Press, 2002, pp. 315-326.

[6] Thomas A. Henzinger, Christoph M. Kirsch, Rupak Majumdar, and Slobodan Matic. "*Time-safety checking for embedded programs*". Proceedings of the Second International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2491, Springer-Verlag, 2002, pp. 76-92.

[7] Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic. "*Schedule carrying code*". Proceedings of the Third International Conference on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2855, Springer-Verlag, 2003, pp. 241-256.

[8] Thomas A. Henzinger and Marco A.A. Sanvido. "*A Programmable Microkernel for Real-Time Systems*" Technical report: UCB//CSD-03-1250, University of California, Berkeley, 2003.

[9] Michael Frisbie, Douglas Niehaus, Venkita Subramonian and Christopher Gill, "*Group Scheduling in Systems Software*" Proceedings of the 12th International Workshop on Parallel and Distributed Real-Time Systems, IPDPS, Santa Fe, New Mexico, April 2004.

[10] Kevin Karplus and Alex Strong, "*Digital Synthesis of Plucked-String and Drum Timbres*", Computer Music Journal, 7(2), Summer 1983, page 43—55

[11] http://www.alsa-project.org/

[12] B. Buchanan, D. Niehaus, G. Dhandapani, R. Menon, S. Sheth, Y. Wijata and S. House "*The Datastream Kernel Interface (Revision A)*", TechReport, dski-1998, Information and Telecommunication Technology Center, University of Kansas, June, 1994, ITTC-FY98-TR-11510-04

[13] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[14] V. Yodaiken and M. Barabanov. "*A real-time linux*". Proceedings of the USENIX conference, 1997. http://rtlinux.cs.nmt.edu/rtlinux/papers/usenix.ps.gz.