Department of Electrical Engineering and Computer Sciences

University of California

Berkeley, California 94720

Visualizing Architecture and Algorithm Interaction in Embedded Systems

Memorandum UCB/ERL M96/50

Farhana Sheikh

September 13, 1996

Research Project

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

Approval for the Report and Comprehensive Examination:

Committee:

Professor E.A. Lee
Research Advisor

Date

Professor T. Henzinger
Second Reader

Date

Abstract

Embedded systems are increasingly becoming integral parts of almost all technology-oriented applications. The complexity and sophisticated nature of these such systems make it very difficult for engineers to exploit the full potential of the system's underlying resources. More often than not, this results in sub-optimal performance. Tools that allow an engineer to quickly evaluate system behaviour and performance can reduce development costs and time-to-market. Visualization techniques have proven invaluable to the design process in the past as they have greatly simplified tasks faced by engineers. Visual representations of views that depict algorithm and architecture interaction are developed to highlight poor algorithm design, problematic hardware-software interfaces, and other reasons behind poor performance in embedded systems. An objected-oriented framework for visual display of design information has been designed and an implementation of this model is discussed. The framework is then used to develop a prototype that implements architecture-algorithm visualization techniques. The theoretical background and issues relating to effective embedded system design are also discussed.

Acknowledgements

I am grateful to my advisor Professor Edward A. Lee for his support and guidance. I would also like to thank John Reekie, Alain Girault, Mike Williamson, and Jose Pino for their invaluable comments and advice. This work was supported in part by the National Science and Engineering Research Council of Canada and I am grateful for their financial backing.

1 Introduction

Over the past decade, there has been a steady increase in the number of applications that demand customized computer systems that offer high performance at low cost. These applications are, more often than not, characterized by the need to process large amounts of data in real time. Examples include consumer electronics, scientific computing, and signal processing systems. A selection of applications is given in Table 1 below.

Constraints on performance, cost and power make software implementations of data processing algorithms for such systems infeasible. Non-programmable hardware, however, does not support modifications of algorithms. The solution to this dilemma has been to develop application-specific hardware that is flexible and programmable – these systems are commonly referred to as embedded systems. They typically include embedded software that is burned into Eraseable Programmable Read Only Memory (EPROM) or resident in memory, special-purpose hardware, and Field Programmable Gate Arrays (FPGAs); often there are stringent requirements on power consumption, performance, and cost. Embedded systems cannot be redesigned or removed easily once the device that incorporates the system has been built.

Embedded systems development thus requires concurrent work on both hardware and software components. The sophisticated nature of the algorithms that are run on these customized computer systems and the complexity of the hardware architecture make it very difficult for engineers to design algorithms that take full advantage of the underlying resources. This often results in sub-optimal performance and under-utilized hardware. Tuning the algorithm for optimum performance can be a very time-consuming and difficult task, especially if the system architecture is complex.

Table 1. *Some applications that incorporate embedded systems*

Military	Communications, radar, sonar, image processing, navigation, missile guidance
Automotive	Engine control, brake control, vibration analysis, cellular telephones, digital radio, air bags, driver navigation systems
Medical	Hearing aids, patient monitoring, ultrasound equipment, image processing, tomography
Telecommunications	Echo cancellation, facsimile, speaker phones, personal communication systems (PCS), video conferencing, packet switching, data encryption, channel multiplexing, adaptive equalization
Consumer	Radar detectors, power tools, digital TV, music synthesizers, toys, video games, telephones, answering machines, personal digital assistants, paging
Industrial	Robotics, numeric control, security access, visual inspection, lathe control, computer aided manufacturing (CAM), noise cancellation

Interfacing software with hardware is a critical issue in embedded system design, since the best performance is achieved when algorithm and architecture interact to reduce communication costs – between hardware components, input or output operations, and memory operations. Tools that provide the engineer with the ability to quickly develop, test, and refine algorithms on different hardware architectures are invaluable to the design process. Time to market and development costs are reduced, costly bugs can be eliminated, possible system failures may be avoided, and a wider set of possible solutions may be explored.

In the past, visualization techniques have been applied to various areas in engineering to simplify tasks faced by a design engineer. These techniques vary from drawing free-body diagrams or circuit diagrams to visualizing scientific data. In all cases, the visual techniques have proven to be invaluable to the design process. An early example of how visualization was used to improve worker performance is given in Gantt's 1919 paper, "Organizing for Work", where he shows how charts – visual representations of machine utilization, distribution of tasks across machines, and worker performance – easily disclose possible reasons for poor worker performance [7].

Recently, system engineers have realized that visualization techniques can be very useful to the embedded-system design process. Poor algorithm design, problematic hardware-software interfaces, and other possible reasons for sub-optimal performance are easily discovered by using such techniques. They have also realized that performance-based design can reduce the time to market a product greatly.

Scheduling is an important part of performance-based design. The scheduling process assigns software tasks to available hardware resources and determines the execution order of the tasks. For instance, a schedule that results in high resource utilization and low communication overhead can greatly improve performance. Even though engineers have realized that scheduling is an extremely important part of the design of an embedded system, they lack adequate tools for exploring appropriate scheduling strategies that exploit the full potential of a target architecture to meet timing and other performance constraints.

This report is divided into four parts. Following this introduction, concepts and issues relating to embedded system design are discussed. The next section builds upon the previous section and describes techniques that can be used to visualize the interaction between a given hardware architecture and an algorithm that is to be run on it. The goal is to easily identify whether the design of the algorithm is well-suited to the underlying hardware architecture of the embedded system or vice versa (the architecture is well-suited to the algorithm). This allows the designer to achieve the best performance possible. It may also provide a means to predict performance at an early stage of design. The third part focuses on the development of an object-oriented model for visual display of design information. The final part describes the software implementation of this model in a prototype that is used within the Ptolemy framework. The prototype implements architecture-algorithm interaction visualization techniques.

2 Background and Theory

This section presents some of the principles and issues concerning the design of embedded systems that motivated the research presented in this report.

2.1 Embedded Systems and Their Design

A *system* can be defined as a group of devices or artificial objects or an organization forming a network especially for distributing something or serving a common purpose [13]. To *embed* a system into some object means to make that system an integral part of the object. When an engineer talks about an embedded system, he or she is usually referring to a system that satisfies a well-defined need at a specific instant in time. The system is usually dedicated to that need, and its operational limits are clearly defined: lifetime, power consumption, performance, and so on. The system usually has limited capabilities for future development, simply because it is permanently installed in a device that provides a certain service to its user. Examples include DSP processors in hand-held communication devices, programmable controllers installed in robots or cars, and video signal processors in television sets.

Because these systems cannot be redesigned or removed easily once the device that incorporates the embedded system is built, the development procedure must produce a correct system that meets all of its operational requirements. In addition, techniques used to design such systems must reduce development costs and time thereby reducing time to market. This is important since introducing a product to the consumer market early can mean that the producer will be facing fewer competitors and hence earning greater revenues and market share.

As stated in the introduction, some of the characteristics of embedded systems include embedded software that is burned into EPROM or resident in memory, special-purpose hardware, FPGAs, stringent requirements on power consumption, performance, and cost. Clearly, an embedded system consists of both hardware and software components. The performance and cost constraints make it necessary for the design engineer to explore a combination of possible hardware architectures or custom hardware components and software or programmable parts that would best suit the nature of the application. Hence, the division between the programmable and non-programmable components and their interface can become a critical issue in the design.

The development process is usually cyclic. The engineer often prototypes an algorithm, tests it on a specific hardware architecture, and then refines the software to make most efficient use of the underlying hardware. If software is to be embedded, this type of development can be very expensive and time-consuming as performance analysis is done after system components have been functionally tested and integrated. This traditional approach to design is shown in Figure 1 and is contrasted to *performance-based design*. Performance-based design advocates evaluating performance at early stages of design [27] such as after functional testing is complete. It is apparent from the figure that the number of iterations required to refine the design after integration of hardware and software would be much less than if the traditional approach was taken since performance is evaluated at a very early stage in development. Clearly, this strategy reduces implementation cost and time-to-market.

Modeling a system and simulating it before actual implementation can further reduce implementation cost and allow the user to explore the design space in search of an optimal solution. Finding this optimal solution can be a complicated task. The designer needs to be able to measure performance and decide whether other operational constraints will be met. The data that needs to be analyzed – processor utilization, number of input or output accesses, number of memory accesses, and inter-processor communication overhead – can be large and difficult to sift through. A good visual display of data can lead the designer to quickly determine whether an algorithm and architecture will perform the given task optimally.

There are very few tools available that allow for performance-based design analysis through visualization as well as analytical techniques [28]. There are even fewer tools that allow a designer to explore embedded software solutions for multiple existing “off-the-shelf” hardware architectures. An important part of fitting software to hardware resources is the scheduling process. This is discussed in the next section.

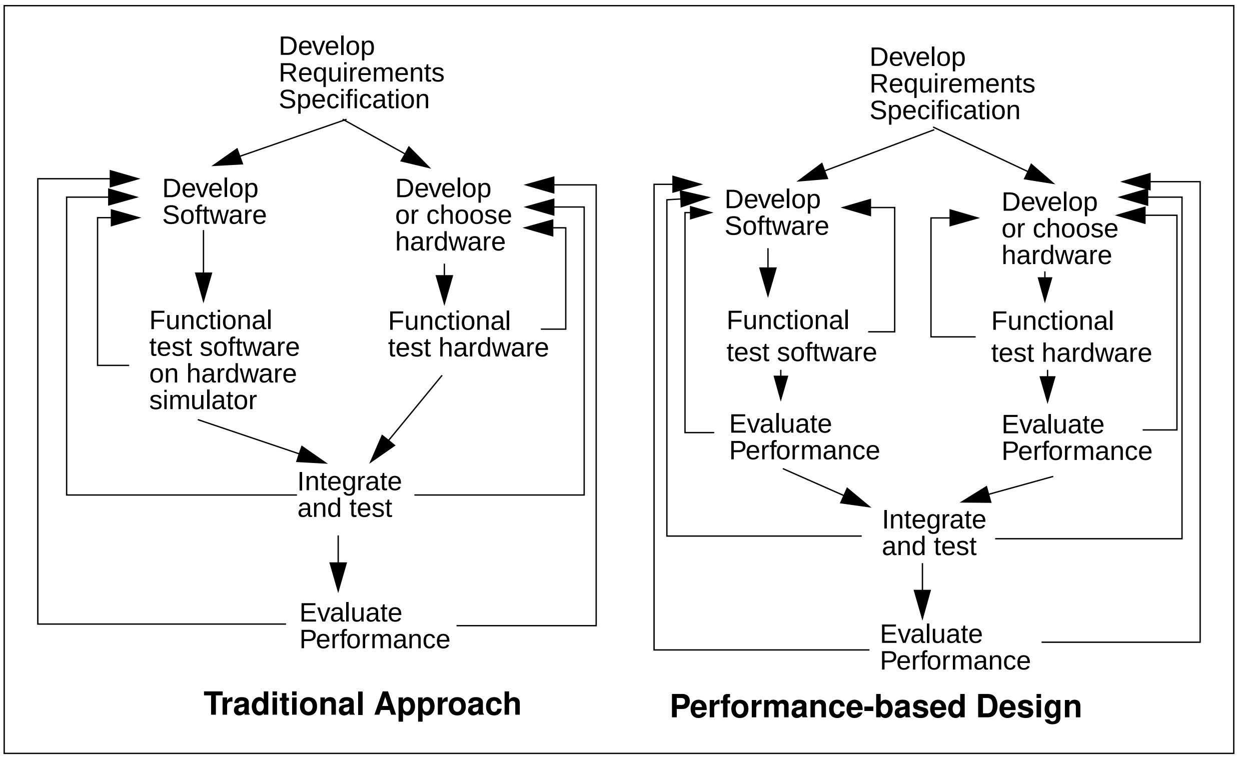


Figure 1. Traditional Design Process Versus Performance-Based Design

2.2 Scheduling Concepts

Scheduling is an important part of the synthesis and operation of any system. The scheduling process assigns a subset of all the tasks that the system must perform to its available resources. The performance of a system can be greatly affected if the tasks are allocated to components that cannot efficiently perform those tasks. The sequence in which the tasks are executed can affect performance also.

When designing embedded systems, the engineer tries to ensure that utilization of hardware resources by software is high at all times. Mapping software to hardware components can be a formidable task if there are a large number of possible hardware resources (e.g. processors) and a large number of ways to partition the software. A number of schedulers, mainly based on heuristics, have been developed to aid the designer map tasks to available processing units and determine their execution order [2][3][23][26].

Scheduling can be performed statically or dynamically. A static scheduler maps tasks to processing units and determines their execution order at compile time, whereas a dynamic scheduler determines this information during run-time. Sometimes choosing one schedule over another may cause a great enhancement in performance. It has not yet been determined whether the reason behind this is that the scheduler is well-suited to the target architecture, or simply because the scheduler does an inadequate job in scheduling the tasks within the given constraints.

Normally, the engineer needs to quickly determine whether the mapping produced by a scheduler will cause the system to perform its intended function within the performance constraints at the lowest possible cost. Traditionally, Gantt Charts, based on concepts developed by Gantt in [7], have been used to visually represent schedules. These charts show how software components are mapped onto processing units over time. A detailed discussion of Gantt Charts is presented in Section 2.4.2.

Current visual displays of scheduling information, which are based on the linear representation developed by Gantt, are only effective in highlighting resource utilization, processor idle time, and task execution order. This representation is inadequate for revealing possible performance bottlenecks and problematic hardware-software interfaces which can greatly affect overall system performance. In addition, scheduling techniques have evolved and a number of the schedulers that use these techniques generate schedules that cannot be effectively displayed using the traditional Gantt Chart. Section 4.3 discusses ways of modifying this display to effectively visualize schedules generated by these newer schedulers.

2.3 Evaluating Algorithms and Their Performance On Given Architectures

It is necessary to gather data about the execution of the algorithm on a target architecture and analyze it in order to evaluate system performance. Evaluating the performance of a system requires gathering, sifting, and displaying many types of information: data from manufacturers, measurements on earlier systems, estimates made during analysis, and availability of logical and physical resources [27]. Scheduling descriptions for logical and physical resources and allocation of processes to hardware or software are emerging to be quite important factors in determining the performance of a system. This section briefly discusses the various types of information that are required to evaluate algorithm performance and current techniques that are used to present the data to design engineers.

- **Processor Utilization**
  Processor utilization is normally determined by finding the time spent computing or doing useful work in relation to the total time required for the task. Traditional Gantt Charts are normally used to display this information.

- **Input or Output Accesses**
  An input or output access occurs when data is either input to the system or output from the system. An input or output operation's performance can be measured in terms of response time or throughput [11]. The response time is measured as the length of time taken for the operation to complete, starting from when the data was placed in a buffer until the time when it is finally output or input. The throughput is the average amount of data that is input or output during a given amount of time.

- **Memory Operations**
  A memory operation occurs when data is either written to or read from a memory location. The memory access time is measured as the hit time plus the miss rate multiplied by the miss penalty (avg.

mem. access time = hit time + miss rate * miss penalty) [11]. It is clear that a memory operation that needs to access data residing in local memory on a processor will take less time than an operation that needs to access data residing at a remote location.

- **Communication Overhead**
  Communication overhead is usually considered important in evaluating multiple processor systems. It is the total time spent sending and receiving data between processors in relation to the total execution time. This information has traditionally been displayed in Gantt Charts. Communication overhead can also be described as the time spent sending and receiving messages between concurrent processes in relation to the overall execution time.
- **Critical Paths**
  A critical path is the longest serial thread, or chain of dependencies, running through an execution of an algorithm [9]. Critical paths are important performance analysis abstractions as the total execution time cannot be reduced without shortening the length of a critical path. Critical paths are potential places for performance bottlenecks.
- **Architecture Bounds**
  There are physical limits placed on all hardware resources such as processor speeds, memory access times, bus speeds, and so on. This of course limits the performance of any software. Each architecture has different bounds, hence a single piece of software may perform differently on different architectures.

2.3.1 Performance and Performance Models

There are number of ways of measuring the performance of a computer system. A system user may evaluate performance by measuring the time that it takes between the start and completion of an event, otherwise known as *response time* or *execution time*, or *latency*. Other people, such as network managers, may use the total amount of work done in a given amount of time which is commonly referred to as *throughput* as a measure of performance. In all cases, time is the basis for measures of system performance [11]. A system that can perform a given task in the least amount of time is said to be the fastest or has the best performance.

Performance models can be useful as they can be used to abstract system behaviour and can allow engineers to predict and analyze system performance. Performance-based design dictates that performance models should be integrated with functional design and resource scheduling. Models that use early estimates of processing costs of parts of a design can aid in planning software architectures, assessing needs to distribute data and functionality, and hardware planning [27].

In order to obtain adequate and predictable performance, it is important to characterize system behaviour: sequences of events, actions, and delays. In addition, it is important to provision and schedule physical and logical resources: hardware (processors, input or output devices, memory, interconnections) and things such as locks and semaphores. Abstract models such as queuing networks, Petri nets, or other simulation models of available resources have been developed in order to aid in performance analysis which helps in determining contention delays, resource saturation at bottlenecks, load imbalance, and interprocessor or interprocess communication overhead [27]. Simulating a system can generate possible execution paths that may reveal critical paths which can lead to discovery of bottlenecks and other causes of low performance.

2.4 Visual Display of Design Information

Design information comes in a multitude of flavours. Whether a system is implemented in hardware or software or both, design information is always present and required at all stages of development. An attempt has been made in Table 2 to categorize the various types in terms of high-level or low-level details whether the design is hardware-based or software-based.

Table 2. *Types of Design Information*

High-Level	**Low-Level**
Specification (requirements, architecture descriptions, interface descriptions)	Implementation (software programs, netlists, hardware description language code)
Documentation (includes bug reports)	Execution history (traces, execution times, latency measurements)
Design constraints (performance, cost, power, memory)	Modification history and version information

For many years, hardware engineers have relied on visual representations of design information to guide them through the development process. Some of the forms that these visualizations take are circuit diagrams, signal traces, and architecture block diagrams. Comparable visualizations for systems implemented in software are few in number. For purely parallel systems, tools that visualize execution, schedules, and communication patterns exist [9] but are sometimes inadequate for complicated architectures. For uniprocessor and multiprocessor embedded systems, especially those that incorporate both hardware and software modules, very little exists in terms of visualizing architecture, execution, and performance.

Recent interest in visualizing software performance analysis of parallel systems has brought about the advent of some basic visualization concepts and principles. These concepts which are briefly summarized in the following section can be used to create effective visual representations of design information.

2.4.1 Visualization Basics

In [9] and [10] M. T. Heath et. al. outline basic concepts and principles that are necessary to produce effective graphical displays. Good visualization techniques can have dramatic impact in areas where they can lead to a discovery of unexpected phenomena. This is something that must always be kept in mind when designing new visualization techniques. Users are not interested in pretty pictures but something that will lead them to construct an empirical model of behaviour.

Some of the basic principles that must be considered when developing visualization techniques are:

- Users should be able to relate the display of information to a context [9]. The visualization should allow a user to connect the display to an environment from which it is derived.
- Any visualization tool must be able to scale easily to large data sets and there must be a means for a user to give the tool feedback.
- A user should be able to tune the tool to their needs.

- An important but often overlooked concept is generation of multiple views of a body of design information. This is important for yielding insight into the behavioural characteristics of the system and their causes [9].
- Techniques such as using colour or size to highlight useful information from large raw data sets are also necessary.

The visualization of algorithm and architecture interaction developed in this project use the above principles as the basis for their design in combination with the principles underlying Gantt Charts, trace displays, and space-time diagrams which are summarized in the following three sections.

2.4.2 Gantt Charts

The Gantt chart was the brainchild of H. L. Gantt, a consulting management engineer who developed methods of planning, production recording, stores-keeping and cost-keeping. He developed three basic charts: Machine Record Chart, Progress Chart, and Man Record Chart. The first displays the amount of time a machine is working, the cumulative working time of an individual machine, the cumulative working time of a group of machines, and reasons for idleness [7]. The Progress Chart gives a distribution of tasks across machines, the rate of work, and the activities on the chart are measured by the amount of time needed to perform them. This chart defines a schedule as it maps tasks to machines and determines their the task execution order. It shows how that schedule is being lived up to by comparing what has been done and what should have been done. If this chart is detailed enough, it can indicate probability of future performance and anticipate needs. The third chart compares what a worker has done with what should have been done: it records a worker's performance. This chart makes it possible to trace the lack of production to its sources.

Gantt created these charts in order to:

- find out how plants are performing the function for which they were created;
- find out reasons why they are not doing as well as they should;
- remove obstacles which hamper them in the performance of their function.

Gantt was trying to solve a planning problem. Designing an embedded system requires similar planning, production recording, stores-keeping, and cost-keeping in order to achieve optimal performance. Hence applying the basic concepts developed by Gantt to embedded system design leads naturally to performance-based design depicted in Figure 1. In addition to using Gantt Charts to discover possible reasons for poor performance, trace displays and space-time diagrams can be useful in empirically modeling system behaviour.

2.4.3 Trace Displays

A trace represents an instance of execution of a system. It helps a user model system behaviour and can lead to discovery of unexpected phenomena. Trace files are used to capture performance details by logging operations performed by a system to a file. This raw data then can be analyzed to determine system behaviour. Good visualizations of traces allow the user to look at the traces in a format similar to looking at signals on an oscilloscope which may reveal critical paths.

2.4.4 Space-Time Diagrams

A space-time diagram shows message passing and communication between different processing units in a system [10]. Diagrams depicting communication behaviour can bring to light patterns of behaviour that can indicate program loops, or can allow a user to determine reasons for low utilization such as poor message-passing techniques and inadequate pipelining.

3 The Ptolemy and Tycho Frameworks

3.1 Ptolemy

Ptolemy is an environment for specifying, simulating, and synthesizing heterogeneous systems. Many of these systems combine control-flow oriented processes with data-flow oriented processes resulting in subsystems that must be modeled using different models of computation. Ptolemy was designed to allow mixing of different models of computation to specify such systems [18]. A system, application, or algorithm can be specified in Ptolemy by representing it visually in terms of whichever semantics seem feasible for the problem. The system specification may contain homogeneous or heterogeneous semantics and a user may wish to divide the system into subsystems that manage specific tasks in a modular fashion. These functional blocks are known as *stars* in Ptolemy. Data that flows along the arcs connecting the stars are known as *particles*. An interconnection of stars is known as a *galaxy* which may represent the entire system or a part of the system. A complete application is known as a *universe* which is an interconnection of stars and galaxies. Hierarchy is used to manage complexity and mix different models of computation [24].

A *target* is a modular object in Ptolemy which describes particular features of the target hardware architecture which will implement the design. It manages a simulation or synthesis process. A user can specify target-specific information either at run-time or choose from a standard set of pre-specified options. If no processor-specific information is provided, the target is asked to determine the communication costs and each of the functional blocks are asked to determine execution time and resources required. The target controls operations such as scheduling, compiling, assembling, and downloading code. A *scheduler* object is associated with each target and it determines how stars will be mapped onto available resources and their execution order. A *domain* is a collection of stars, schedulers, and targets. It implements a particular model of computation [24] and either performs simulation or code generation. Figure 2 clarifies the aforementioned terms and Table 3 [24] summarizes all the domains available within Ptolemy.

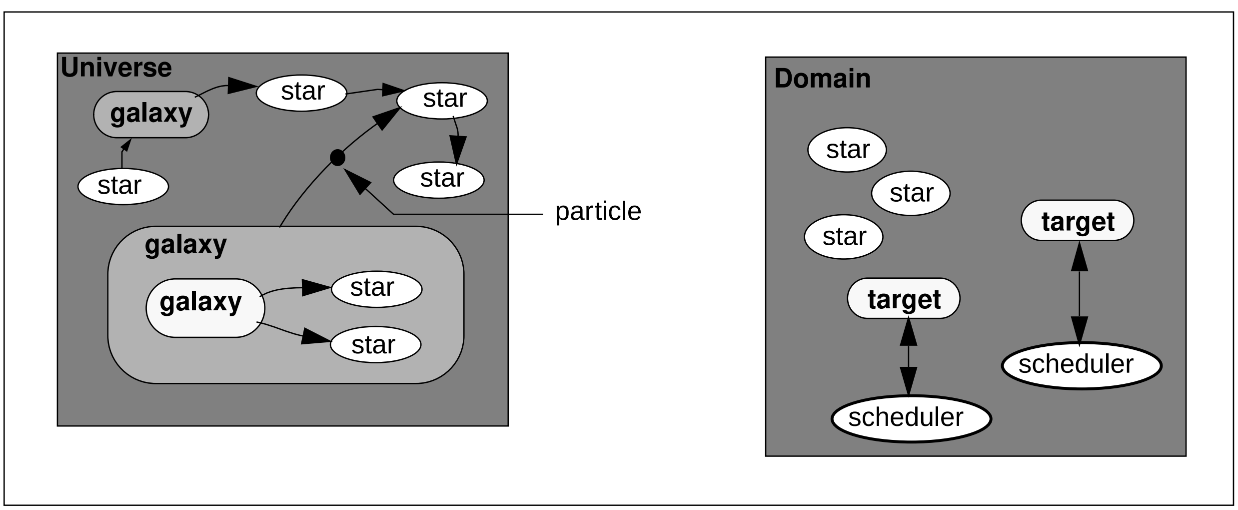


Figure 2. Ptolemy Terminology

Table 3. *Summary of Domains in Ptolemy*

Domain	Description
Synchronous Data Flow (SDF)	• Oldest and most mature domain; it is a sub-domain of DDF, BDF, and PN. • Special case of data flow model of computation developed by Dennis [5]. • Flow is completely predictable at compile time thus allows for efficient scheduling. • Allows for static scheduling. • Good match for synchronous signal processing systems with sample rates that are rational multiples of one another. • Supports multi-rate applications and has a rich star library. • Range of applications is limited.
Dynamic Data Flow (DDF)	• Versatile model of computation as it supports conditionals, data-dependent iteration, and true recursion. • More general than SDF. • Uses dynamic (run-time) scheduling which is more expensive than static scheduling. • Good match for signal processing applications with a limited amount of run-time control.
Boolean Data Flow (BDF)	• Relatively new domain which supports run-time flow of control. • Attempts to construct a compile-time schedule to try and achieve efficiency of SDF with generality of DDF. • More limited than DDF. • Constructs an annotated schedule: execution of a task is annotated with a boolean condition.
Integer and State Controlled Data Flow (STDF)	• Very new to Ptolemy and still experimental. • Realizes data flow control by integer control data and port statuses. It is an extension to BDF. • Scheduling is static and conditional like BDF. • It has user-defined evaluation functions.
Discrete Event (DE)	• Relatively mature domain which uses an event-driven model of computation. • Particles carry time-stamps which represent events that occur at arbitrary points in simulated time. • Events are processed in chronological order.
Finite State Machine (FSM)	• Very new to Ptolemy and still experimental. • Good match for control-oriented systems like real-time process controllers. • Uses a directed node-and-arc graph called a state transition diagram (STD) to describe the FSM.

Table 3. Summary of Domains in Ptolemy

Domain	Description
Higher Order Functions (HOF)	• Implements behaviour of functions that may take a function as an argument and return a function. • HOF collection of stars may be used in all other domains. • Intended to be included only as a sub-domain by other domains.
Process Network (PN)	• Relatively new domain that implements Kahn process networks which is a generalization of data flow – processes replace actors. • Implements concurrent processes but without a model of time. • Uses POSIX threads. • SDF, BDF, and DDF are sub-domains of PN.
Multidimensional Synchronous Data Flow (MDSDF)	• Relatively new and experimental. • Extends SDF to multidimensional streams. • Provides ability to express a greater variety of dataflow schedules in a graphically compact way. • Currently only implements a two-dimensional stream.
Synchronous/Reactive (SR)	• Very new to Ptolemy and still experimental. • Implements model of computation based on model of time used in Esterel. • Good match for specifying discrete reactive controllers.
Code Generation (CG)	• Base domain from which all code generation domains are derived. • Supports a dataflow model that is equivalent to BDF and SDF semantics. • This domain only generates comments, allows viewing of the generated comments, and displays a Gantt Chart for parallel schedules. • Can only support scalar data types on the input and output ports. • All derived domains obey SDF semantics. • Useful for testing and debugging schedulers. • Targets include bdf-CGC which supports BDF, default-CGC which supports SDF semantics, TclTk_Target which supports SDF and must be used when Tcl/Tk stars are present, and unixMulti_C which supports SDF semantics and partitions the graph for multiple workstations on a network.
Code Generation in C (CGC)	• Uses data flow semantics and generates C code. • Generated C code is statically scheduled and memory used to buffer data between stars is statically allocated.
Code Generation for the Motorola DSP 56000 (CG56)	• Synthesizes assembly code for the Motorola DSP56000 family.

Table 3. Summary of Domains in Ptolemy

Domain	Description
Code Generation in VHDL (VHDL, VHDLB)	• Relatively new and experimental • Generates VHDL code. • VHDL domain supports SDF semantics whereas VHDLB supports behavioural models using native VHDL discrete event model of computation. • Many targets to choose from. • VHDL domain is good for modeling systems at functional block level whereas VHDLB is good for modeling behaviour of components and their interactions at all levels of abstraction.

A user may simulate the specified system or generate code for a specific target architecture. Once a user selects a target architecture, code generation can begin. Scheduling is the first stage of code generation in Ptolemy and Section 3.3 discusses it in detail

Developers and users of Ptolemy have found that it has lacked proper facilities to visualize execution traces, schedules for uniprocessor and multiprocessor systems, and performance data. This project aims to fill this need by incorporating appropriate visualization techniques into Ptolemy using Tycho as the basis for the implementation. Tycho is briefly described in the next section.

3.2 Tycho

Tycho is a software system designed to complement the Ptolemy framework with a hierarchal syntax manager. Some of the key objectives of the Tycho project are to provide an extensible framework for experimentation with visual syntaxes and to provide a mechanism for system design management. Tycho is based on an object-oriented software architecture which is designed to allow for easy integration of textual and graphical editors and displays. The focus is on allowing users to mix different syntactic models such as allowing combinations of textual and graphical syntaxes. It is a relatively new project and is still in its infancy stages.

3.3 Scheduling in Ptolemy

As stated earlier, scheduling is the first stage of code generation in Ptolemy. Currently, code generation facilities exist for Synchronous Data Flow (SDF), Boolean Data Flow (BDF), and Integer and State-controlled Data Flow (STDF) semantics [24]. Since all of the code generation domains support SDF semantics and due to the fact that the SDF domain is the most mature of all the domains, the following description of scheduling will be discussed with respect to SDF semantics. The concepts are easily extended to the other domains.

Code generation begins after the application or algorithm has been specified using a data flow graph and a target hardware architecture description has been selected. The target object contains information specific to the hardware architecture: number of processors, communication costs, interconnection topology, and so on. Several single and multiple processor schedulers use different algorithms for determining partitioning and order of execution of functional blocks. For multiprocessor systems, an acyclic precedence graph (APEG) must be created for every SDF graph before a schedule can be generated. The APEG displays the precedence relations between the invocations of the SDF functional blocks. All schedulers designed for multiprocessor systems use the generated APEG as input [24].

Several types of schedulers exist within the Ptolemy framework. Some schedulers ignore hierarchy that may be present in the SDF graph in order to maximize concurrency, whereas others use hierarchy to minimize complexity. There are some schedulers that have been designed by Ptolemy researchers that create a new hierarchy by clustering a scheduling graph to take advantage of the natural looping structure of the code. No single scheduler can handle all situations so Ptolemy allows a user to mix and match different schedulers for specific applications [24].

A summary of the types of schedulers available in Ptolemy is given in Table 4 and Table 5 [24]. The terminology used in the tables is described in the following section.

3.3.1 Scheduling Terminology

This section briefly describes some of the terms used in Table 4 and Table 5.

- **APEG**
  APEG abbreviates acyclic precedence graph. The nodes in this graph represent tasks or computations and the directed arcs represent precedence constraints and data paths. Each arc has a label which specifies the amount of data that the source node passes to the destination node. An APEG can be derived from a SDF representation [26].
- **Looped Schedule**
  A *looped schedule* is one which may contain any number of *schedule loops*. A schedule loop consists of m number of actors or functional blocks that are repeated in succession n times. Loops in a looped schedule may be nested [3].
- **Clustering**
  For each node in a SDF graph, there are q corresponding nodes in an APEG. The number q represents how many times the SDF node must be invoked in order to satisfy data precedences in the SDF graph. This expansion can result in exponential growth of nodes in the APEG. *Clustering* SDF graph nodes into composite nodes can limit the expansion resulting in a simpler APEG. The clusters may be scheduled much like actors resulting in hierarchical schedules [22].
- **Buffers and Buffering**
  Each edge in a SDF graph corresponds to first-in-first-out queue that buffers tokens that pass along the edge. The queue is known as the *buffer* for the edge; the process of maintaining the queue of tokens is known as *buffering* [3].
- **Single Appearance Schedule**
  A *single appearance schedule* is one in which each actor or functional block appears only once in the entire schedule [3].

Table 4. *Single Processor Schedulers in Ptolemy*

Scheduler Name	Features
Default SDF Scheduler	• Performed at compile time. • Many possible schedules but schedule is chosen based on a heuristic that minimizes resource costs and amount of buffering required. • No looping employed so if there are large sample rate changes, size of generated code is large.
Joe's Scheduler	• Performed at compile time. • Sample rates are merged wherever deadlock does not occur. • Loops introduced to match the sample rates. • Results in hierarchical clustering. • Heuristic solution so some looping possibilities are undetected.
SJS (Shuvra-Joe-Soonhoi) Scheduler	• Performed at compile time. • Uses Joe's Scheduler at front end and then uses an algorithm on the remaining graph to find the maximum amount of looping available.
Acyclic Loop Scheduler	• Performed at compile time. • Constructs a single appearance schedule that minimizes amount of buffering required. • Only intended for acyclic dataflow graphs.

Table 5. *Multiple Processor Schedulers in Ptolemy*

Scheduler Name	Features
Hu's Level-based List Scheduler	• Performed at compile time. • Most widely used. • Tasks assigned priorities and placed in a list in order of decreasing priority. • Ignores communication costs when assigning functional blocks to processors.
Sih's Dynamic Level Scheduler	• Performed at compile-time. • Assumes that communication and computation can be overlapped. • Accounts for interprocessor communication overheads and interconnection topology.

Table 5. *Multiple Processor Schedulers in Ptolemy*

Scheduler Name	**Features**
Sih's Declustering Scheduler	• Performed at compile-time. • Addresses trade-off between exploiting parallelism and interprocessor communication overheads. • Analyzes a schedule and finds the most promising placements of APEG nodes. • Not single pass but takes an iterative approach.
Pino's Hierarchical Scheduler	• Performed at compile time. • Partially expands the APEG. • Can use any of the above parallel schedulers as a top-level scheduler. • Supports user-specified clustering. • Realizes multiple orders of magnitude speedup in scheduling time and reduction in memory usage.

3.3.2 Schedule Files

All scheduler objects in Ptolemy have a method that will allow a user to display a schedule in text form. The format of the string is constructed to allow visualization tools to easily parse the schedule information and construct a visual representation of the schedule. The format of the string is displayed below in Table 6.

The notation used in Table 6 is explained below:

- (<*name*>) + means one or more items of type *name*;
- <type:description> means an item of the given type and description;
- { and } are included to make the string trivial to parse in Tcl;
- bold-faced words are key words indicating what type of information will follow.

All entries are optional and as more schedulers are added to Ptolemy, the list is expected to grow. Some items are used only by specific schedulers. For example, the *assign* item is used by a BDF scheduler to record a value that affects the schedule.

A schedule file contains a <*schedule*> string that contains one or more entries of the type <*entry*> which can be many different items as shown in the table. It can be either a nested schedule, a numerical entry indicating either performance or schedule data, or it can be an identifier. Some examples of schedule files can be found in Table 7. Visualization representations of schedule files is discussed in the next section.

Table 6. *Schedule File Format*

<table>
<tr><th>Entry</th><th>Format</th></tr>
<tr><td><<i>schedule</i>></td><td>{ (<<i>entry</i>>)+ }</td></tr>
<tr><td><<i>entry</i>></td><td>{ scheduler <string:scheduler_identifier> }
| { galaxy <string:galaxy_or_universe_name> }
| { target <string:target_name> }
| { (<<i>num</i>>)+ }
| { utilization <float:utilization> }
| { (<<i>nestedSchedule</i>>)+}
| { cluster <string:name> <<i>schedule</i>> }
| { assign <string:token> <string:value> }
| { fire <<i>star</i>> }
| { fire <<i>star</i>> <<i>f_info</i>> }
| { processor <string:name> <<i>schedule</i>> }
| { repeat <int:repetitions> <<i>schedule</i>> }
| { <string:annotation> <<i>schedule</i>> <string:endannotation>}</td></tr>
<tr><td><<i>nestedSchedule</i>></td><td>{ preamble <<i>schedule</i>> }
| { cluster <<i>schedule</i>> }
| { <string:annotation> <<i>schedule</i>> }</td></tr>
<tr><td><<i>num</i>></td><td>{ numberOfProcessors <int:numprocs> }
| { numberOfStars <int:numstars> }
| { numberOfStarOrClusterFirings <int:sizeofDAG> }
| { makespan <int:makespan> }
| { totalIdleTime <int: idletime> }
| { idle <int:idletime> }</td></tr>
<tr><td><<i>star</i>></td><td>send
| receive
| <string:star_name></td></tr>
<tr><td><<i>f_info</i>></td><td>{ (<<i>f_entry</i>>)+ }</td></tr>
<tr><td><<i>f_entry</i>></td><td>{ exec_time <int:exec_time> }
| { start_row <int:start_row_index> }
| { start_col <int:start_col_index> }
| { end_row <int:end_row_index> }
| { end_col <int:end_col_index> }
| { <string:label> <string:value> }</td></tr>
</table>

4 Techniques to Visualize Algorithm and Architecture Interaction

This section presents techniques that can be used to visualize the interaction between an embedded system architecture and algorithms that are run on it. The concepts that were summarized in Section 2 are used as the basis for the design of a tool that allows an engineer to visualize the interaction and performance of various components in an embedded system. Details of the software architecture and implementation are the subjects of the final two parts of this project.

The initial goals are to be able to animate execution of algorithms on selected targets and to be able to easily identify performance bottlenecks, load imbalances, and problematic hardware-software interfaces. This requires designing visual representations of embedded system hardware and software components, schedules, execution traces, communication patterns, and performance data. All the representations described in this section are discussed with respect to Tycho and Ptolemy, however, they are not necessarily tied to these frameworks.

The first subsection that follows describes visual representation of hardware components of an embedded system. Following sections describe visualization of various types of schedules, execution traces, and communication patterns. The final subsection discusses what sort of extra insight can be gained about system behaviour by merely combining these views into a single display.

4.1 Visual Representation of Hardware Components

As stated in the discussion of visualization basics, it is important for any visualization tool to allow a user to relate displayed design information to a context. In the case of embedded system design, an engineer would like to relate visual representations of execution and performance data to the system under analysis. A visual representation of system components and how they are being employed provides a way of relating execution behaviour to the system environment. For example, if a visual representation of the hardware architecture and its bounds are available to the engineer, he or she can quickly determine whether poor performance is due to the architecture bounds or due to poor software design or a combination of both.

A simple yet effective method of representing an architecture visually is shown in Figure 3 below. Each large block represents a processing unit. A processing unit may be an off-the-shelf processor or a memory management unit, or a FPGA, or even an application specific integrated circuit (ASIC). The amount that each block is filled indicates how well that resource is being utilized. If the block is completely filled then the resource is fully employed in executing tasks at all times.

Interconnections between processing units is indicated by solid lines. Dashed lines indicate that a connection is possible however, in the depicted configuration, no connection has been made. The benefit of indicating different architecture configurations and delineating them from what is actually being used in the current setup reminds the user that other design possibilities exist which may achieve better performance. In addition it may allow a designer to gain insight into system behaviour by changing configurations quickly and observing changes in execution patterns and performance. The thickness of each of the arcs connecting the processing units indicates the relative amount of communication taking place between the units.

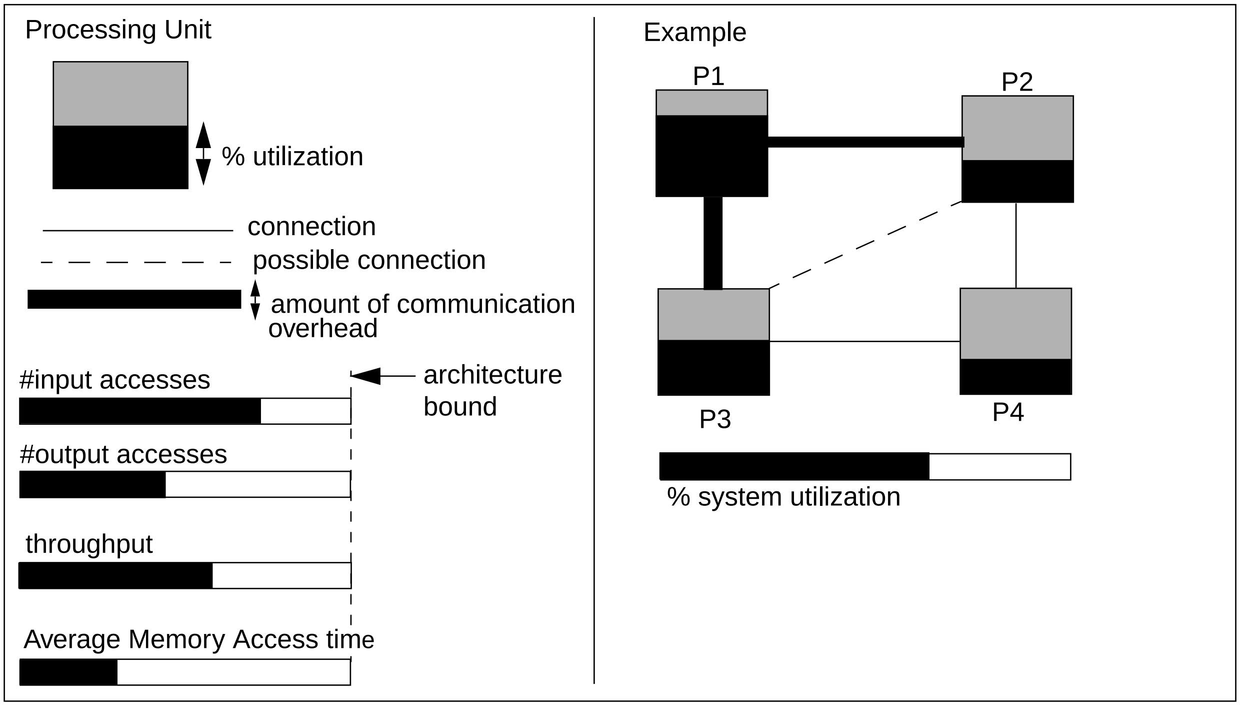


Figure 3. Visual Representation of Hardware Architectures

Clicking on an individual processing unit block bring up information regarding bounds inherent in the resource such as speed, and size of local memory. In addition, statistical information such as the number of input or output accesses, throughput, and average memory access times are also displayed. These statistics are shown in a manner that allow a designer to compare the observed throughput or average memory access time with what is actually possible for that resource. This is useful in indicating to the designer how well the architecture is being employed in comparison to its full potential and helps in determining whether the system projected onto the hardware should be redesigned or whether the hardware should be changed to suit the system needs.

The visualization described above allows an engineer to quickly determine how the interconnection topology is being used and at glance he or she can determine utilization and communication overhead. If the designer wants to dig deeper and get more detail, clicking on each individual block representing a processing unit will bring up relevant information. Layering information in this manner allows for uncluttered displays and allows a user to tune the tool his or her needs.

4.2 Visual Representation of Software Components

Ptolemy provides a means to create an abstract functional model of an entire system. However, the system may be implemented in either hardware or software or a combination of both. In the previous section, visual representation of hardware components was discussed and it was shown how this was useful in gaining insight into system behaviour and providing a means for relating performance data and execution traces to the system environment. Due to the fact that not all embedded systems are implemented solely in hardware, it is not enough to only provide a user with a hardware context. A representation of software architectures equivalent to hardware representations would be extremely useful in depicting interconnection and interaction of software components. It would also serve as an effective way of displaying software designs. This work is beyond the scope of this project, however, developers of Tycho may envision adding this capability at a later time.

4.3 Visual Representation of Schedules

The Gantt Chart discussed in Section 2.4.2 has traditionally been used to display schedules, specifically in the form of the Progress Chart. The other two charts – Machine Record Chart and Man Record Chart – have been used to a much lesser extent as a basis for charts that show schedules or other performance data. In attempting to create an effective display of schedules generated for uniprocessor and multiprocessor systems by Ptolemy schedulers, the following questions were asked:

- How are various components of the embedded system performing the function for which they were created?
- Is it possible to identify obstacles that hamper performance of the various components?
- Are all resources being used effectively? If not, why not?
- If all components are busy, are they executing tasks that have high priority? If not, why not?
- If all components are busy and executing high priority tasks, are they doing it as fast as possible? If not, why not?

Clearly large amounts of data are required to answer the above questions and it is not possible to display all the data in a single view. This is where it is necessary to employ the concept of multiple views. Combining these views in a single display can provide extra insight into behaviour and possible reasons behind poor performance that would not be gained if the views were seen separately. This is discussed in detail in Section 4.6.

A visual representation of the chosen hardware architecture constitutes the first view which provides a description of the system environment and at a glance gives an indication of performance. This was discussed in the preceding section and it is similar to the Machine Record Chart concept discussed in Section 2.4.2. It tries to give clues to the designer regarding possible reasons for poor utilization, and why some resources may not be effectively employed in performing tasks.

The Progress Chart concept is used as the basis for a view that displays scheduling information. It shows the distribution of functional blocks over available resources. The chart indicates the number of processors, the distribution of tasks across the processors, times when the processors are idle or are busy sending and receiving messages from connected processors. It does not show communication patterns but does indicate utilization. The chart created for Ptolemy schedules is different from traditional Gantt Charts in that the display is not strictly linear. Because Ptolemy allows mixing of schedulers which sometimes results in hierarchical scheduling, it was necessary to create a display that would indicate this mixing and the resulting hierarchy.

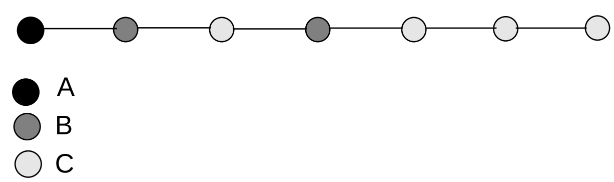
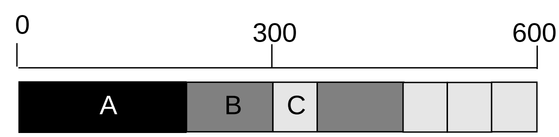
The various forms that a schedule can take in Ptolemy are shown in Table 7. The simplest form is a sequence of firings of actors or functional blocks on a single processor. Due to the fact that no execution times are associated with the actor firings, the sequence is depicted as a train of coloured circles, each colour representing a different actor. If execution times are associated with each actor then a strip of coloured blocks represents the firing sequence. The length of each block gives an indication of relative time spent executing that particular task. The time line at the top of the view indicates the total length of the schedule and processor utilization is given below the schedule.

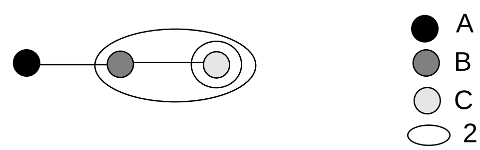
Loops in a schedule are shown in the third example in Table 7. If a sequence of actors occurs within a schedule loop, then an ellipse with a certain thickness is drawn around the actors to indicate the looping. The legend on the side provides a key to which colours correspond to which actors and how many iterations of the schedule loop the thickness of the ellipse represents. Even though this is a very simple visualization, it compactly displays looping and loop nesting, and order of execution.

Hierarchy which can occur because of clustering is indicated by a box that has a raised relief and a special cluster icon. It can be opened up by clicking on it with a mouse to reveal a nested schedule which is shown in a separate display. The nested schedule may be a looped schedule or another cluster or any other type of schedule.

For multiprocessor schedules, communication between processors is shown by send and receive blocks that are indicated by blocks that have special icons. The length of the blocks indicate how much time is required by each send and receive. Displaying communication on a schedule allows a user to gauge how much time is spent in communication overhead versus time spent performing useful tasks. If the schedule shows that too much time is being spent in sending and receiving messages, then either the architecture topology is not well-suited to the application or the algorithm is not well-suited to the architecture.

Table 7. *Visual Representation of Different Schedules*

Schedule Example	**Visual Representation**
Simple Sequence of actor firings with no execution times: {**scheduler** “Identifier”} {**fire** A} {**fire** B} {**fire** C} {**fire** B} {**fire** C} {**fire** C} {**fire** C}	
Sequence of actor firings with execution times: {**scheduler** “Identifier”} {**fire** A {**exec_time** 200}} {**fire** B {**exec_time** 100}} {**fire** C {**exec_time** 50}} {**fire** B {**exec_time** 100} {**fire** C {**exec_time** 50}} {**fire** C {**exec_time** 50}} {**fire** C {**exec_time** 50}}	

Looped Schedules – Example 1:
{**fire** A}
{**repeat** 2
    {**fire** B}
    {**repeat** 2
        {**fire** C}
    }
}
	

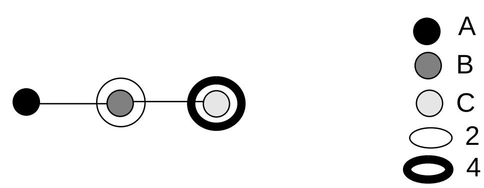
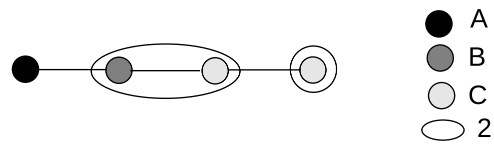
Looped Schedules – Example 2:
{**fire** A}
{**repeat** 2
    {**fire** B}
}
{**repeat** 4
    {**fire** C}
}
	

Table 7. *Visual Representation of Different Schedules*

Schedule Example	Visual Representation
Looped Schedules – Example 3: {**fire** A} {**repeat** 2 {**fire** B} {**fire** C} } {**repeat** 2 {**fire** C} }	

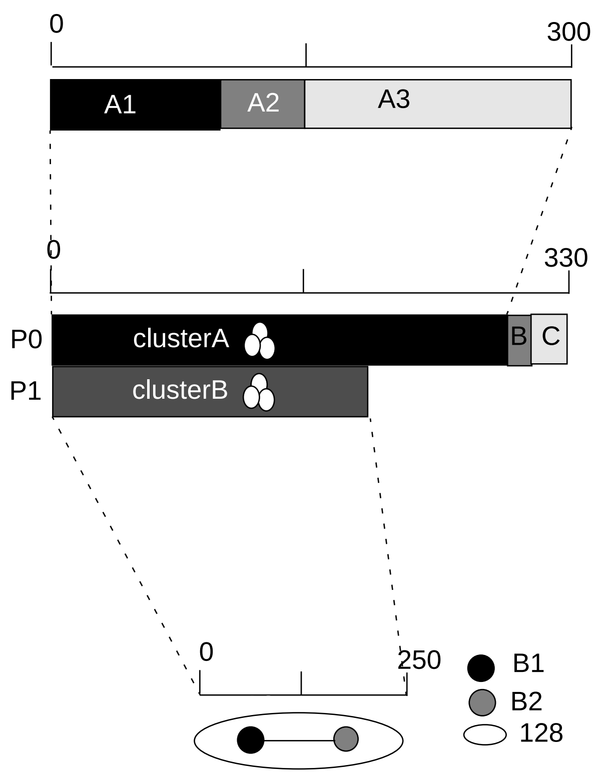
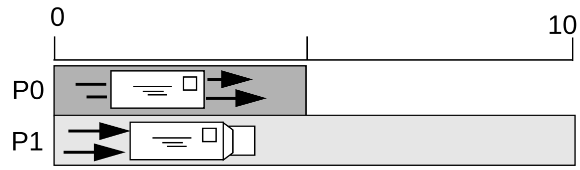
Clusters:
{**cluster** {
{**scheduler** “Identifier”}
{**galaxy** *galName*}
{**numberOfProcessors** 2}
 	
{**processor** 0 {
{**target** target1}
{**totalIdleTime** 0}
{**fire** ClusterA {**exec_time** 300}}
{**fire** B {**exec_time** 10}}
{**fire** C {**exec_time** 20}}
}}
 	
{**processor** 1 {
{**target** target2}
{**totalIdleTime** 80}
{**fire** ClusterB {**exec_time** 250}}
}}
 	
{**cluster** ClusterA {
{**fire** A1 {**exec_time** 100}}
{**fire** A2 {**exec_time** 50}}
{**fire** A3 {**exec_time** 150}}
}}
 	
{**cluster** ClusterB {
{**repeat** 128
{**fire** B1}
{**fire** B2}
}}}
 	
}}
	

Table 7. *Visual Representation of Different Schedules*

Schedule Example	Visual Representation

{**processor** 0 {
{**fire send** {**exec_time** 5}}
}}
 	
{**processor** 1 {
{**fire receive** {**exec_time** 10}}
}}
	

4.3.1 Modifying Schedules

Most of the schedulers in Ptolemy are based on heuristics and do not give point solutions, therefore it is beneficial to allow users to edit a schedule so that the system can be optimized for whichever measure of performance the engineer feels is important for the application. This added flexibility gives users more control over the design of the system and gives them the ability to explore effects of slight schedule modifications on performance. It can also help in providing insight into the scheduling algorithm being employed.

Editable schedules only make sense for those that are static or quasi-static. These schedules are generated at compile time according to some heuristic. Dynamic schedules are generated at run-time hence they cannot be modified before the system executes. The static schedulers listed in Table 4 and Table 5 all consider the data precedences inherent in the data flow specification of the application. Hence, when a user modifies a schedule he or she must not be allowed to make changes that would cause a deadlock in the data flow graph. It is clear that the schedule editor must employ some sort of validation criteria and prevent the user from making inappropriate changes to the schedule. In addition, if a change is made to the schedule in a particular place, it may cause changes to occur in other places. Of course, the editor should have the capability of reflecting the changes caused by these dependencies.

An outline of how back annotation of schedules could be implemented within the software framework developed in this project is discussed in Section 6.8.4.

4.4 Visual Representation of Execution Traces

In Ptolemy it is possible to output an execution trace of any algorithm from any domain. From this a user can create a trace file which like a schedule file can be displayed in graphical form. Visual representation of execution behaviour can reveal critical paths and help determine execution patterns [1]. Much like an oscilloscope records the evolution of a signal over time, the execution trace tracks system behaviour over time. It keeps a record of what tasks were executed, when they were executed, and how long they took to complete. The trace may also be annotated with expected or estimated behaviour. Clearly, this is very similar to the Man Record Chart concept developed by Gantt [7].

A simple visualization of an execution trace is shown in Figure 4. Each task is listed along the vertical axis and time is given along the horizontal axis. The thick lines represent how long each task was expected to take and the thin lines below represent actual execution. The different colours represent different processing units. This simple visualization gives a very good picture of execution behaviour and how it compares to what a designer may have expected or estimated.

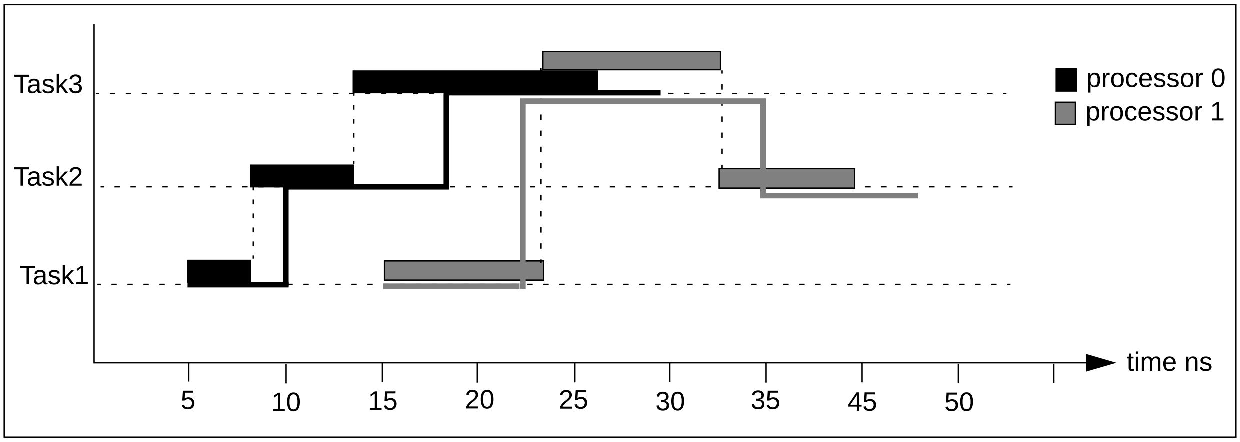


Figure 4. Visual Representation of An Execution Trace

4.5 Visual Representation of Interprocessor Communication

For most parallel systems it is extremely useful to be able to discern communication patterns as they can disclose important properties of algorithms or poor message-passing techniques which would result in poor performance. Schedule files in Ptolemy contain information regarding communication in the form of times taken to send and receive messages however, source and destination information is missing. The string representing schedule information will be modified to incorporate source and destination information. The modifications are discussed in detail in Section 6.10.

The visualization of communication is shown in the figure below. Processing units are shown along the vertical axis and time is given along the horizontal axis. Lines drawn from one processing element to another indicate message-passing between the two and the length of the horizontal lines at each processing unit indicate how much time that processing element is spending executing some task. The time spent sending and receiving messages is shown by the space between when a processor finishes executing its task and when the unit receiving the message starts executing its task. Parallelism can be depicted by overlaying all on-going communication in the same view.

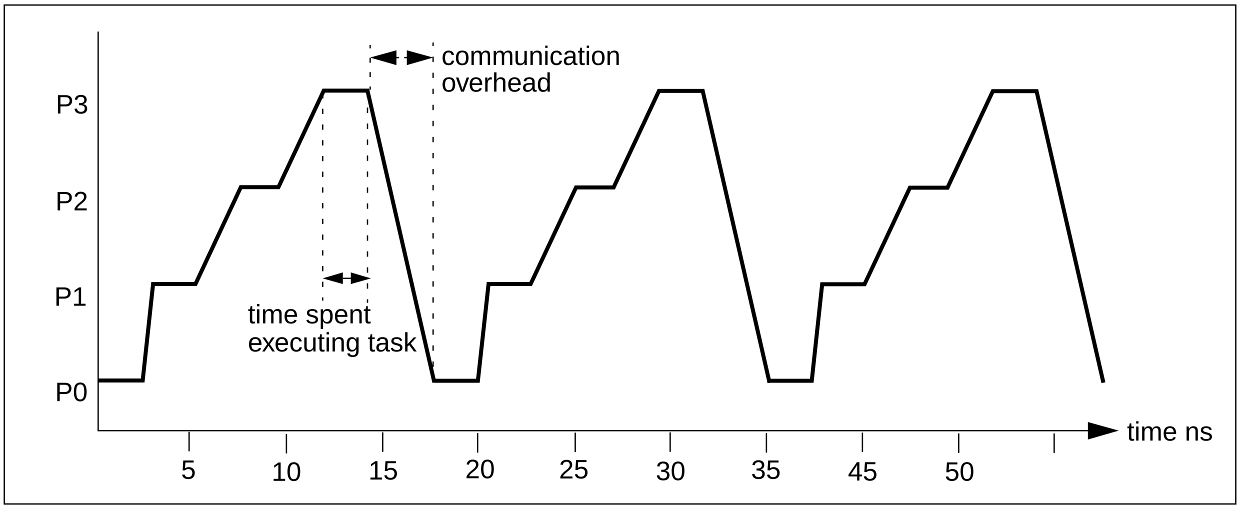


Figure 5. Visual Representation of Communication Between Interconnected Processors

4.6 Combining Views

Multiple views of design information can provide extra insight into system behaviour. Combining these views into a single display can be extremely useful in gaining important information that may not have been gained otherwise. For example, if a view of a schedule and a view of the communication patterns is combined into a single display with one view above the other, then the designer is able to determine how the processors are communicating with one another and how much time they are taking sending and receiving messages in relation to the time taken to execute other tasks. Another useful combination is that of an architecture view and a view of a schedule. The engineer can almost instantly relate scheduling information to a context and gain insight into how topology can effect task execution and performance. This may also helps in making changes to a schedule that would make better use of the underlying resources.

5 An Object-Oriented Model for Visual Display of Design Information

This section presents an object-oriented model for visual display of design information. The model consists of three key objects that allow for combinations of multiple views of design information: Displayer, View, and Data Filter. Each of these objects are described in detail in the following subsections. The notation used in the figures describing the object-oriented architecture is described in [25] and summarized in the figure below.

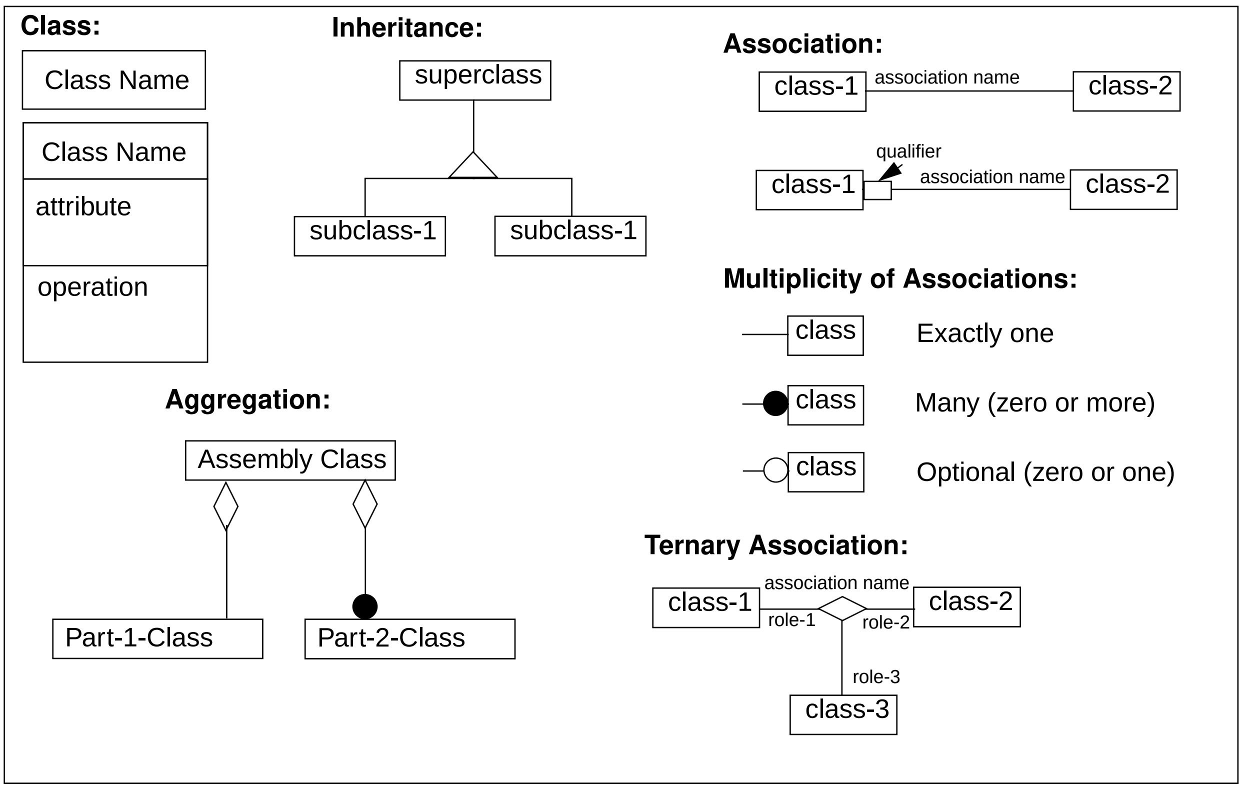


Figure 6. Summary of Object Model Notation

An overview of the design is given in Figure 7. Two key issues that motivated the design are:

- Different views of information need to be easily combined in a single display. This is useful in providing insight as a single view may not present enough information to the designer to construct a proper empirical model of system behaviour. Combining multiple views of design information in a single display can sometimes give a user more information than if the views were seen separately. The benefits of this with regards to algorithm and architecture interaction were discussed in Section 4.6.
- If one set of design information is being used in multiple views, then the current *information state* needs to be preserved across views. In other words, if a piece of design information is being edited in one view, then the changes must be reflected in the views that may be displaying the same piece of

information in a different format. In addition, it must be possible to save changes to the design information in the format that it was originally stored in. For example, if a text file was rendered in graphical form in a view and changes were made to the view graphically, then it should be possible to save those changes in a text form that may be re-read and displayed in another form by another view later on in the design process.

As one can see from the figure below, the Displayer can house zero or more views and in turn, the view can include sub-views. This makes it possible to combine multiple views in a single display and combine various syntaxes (e.g. text and graphics) in a single view.

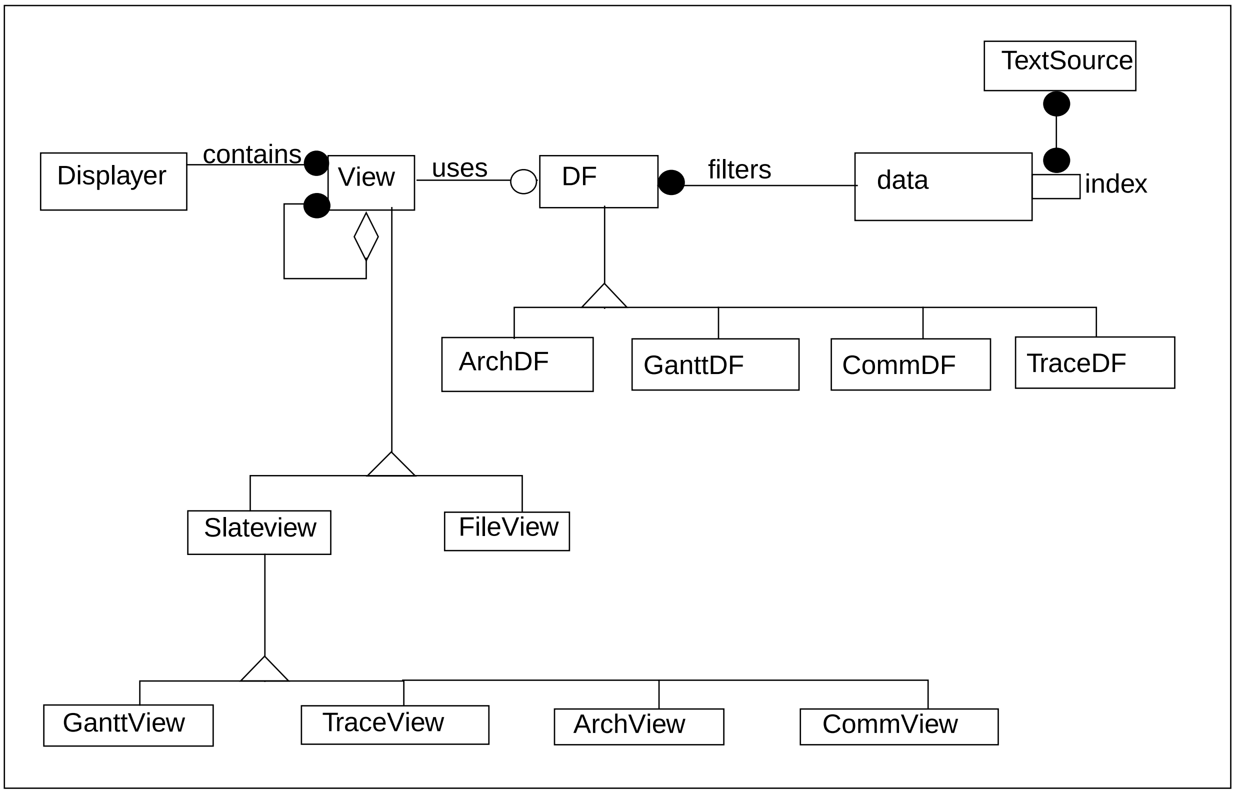


Figure 7. Overview of the Object-Oriented Model for Visual Display of Design Information

The Displayer is responsible for inserting and removing Views. It also responsible for assigning menuberas to registered Views. The Displayer is discussed in detail in Section 5.1. The main purpose of a View is to render design information. It can choose a Data Filter (DF) which will present the View with a sequence of display data that indicates how the information should be displayed. The View can query the Data Filter for any key parameters in addition to asking for the display data at any moment in time.

The Data Filter transforms data presented to it from one form into another. For example, it may take a SDF data flow graph representation and generate an APEG representation. It may also take a text string, strip out important information, and then generate another string consisting of graphical annotations that a View can use to graphically display information contained within the original text string. The View and the Data Filter are discussed in Section 5.2 and Section 5.3, respectively. Both View and Data Filter are abstract objects that other objects can inherit from.

5.1 Display Management

The Displayer class manages the display of views and menus. Views can be inserted or removed from the Display using methods such as *setView* or *removeView*. It is also responsible for providing Views with a menubar for their use. Each View can configure its own menubar by employing the menu bar methods and once complete, the View need only tell the Displayer that it now has the focus and would like it to Display its menubar.

Even though the Displayer can house any number and type of Views, it is assumed that users of the Displayer will practise good judgement when mixing Views. If one or more radically different Views are mixed, then the user does not gain anything from putting the Views into a single display. The main purpose of mixing Views is to gain insight that would not have been obtained otherwise. Details of the Displayer implementation are discussed in Section 6.2 and a brief summary of the intended functionality is discussed in the sections immediately following this one.

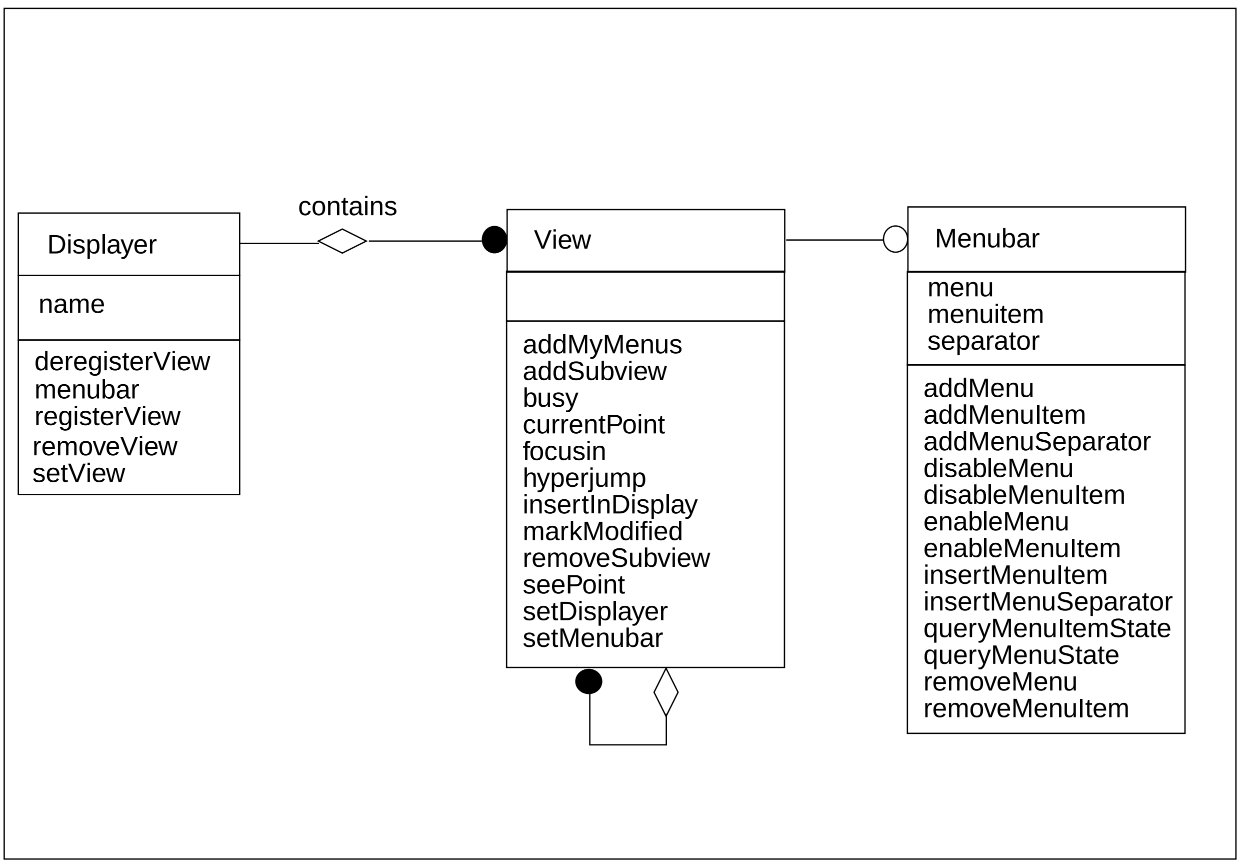


Figure 8. The Displayer, Menubar and View Objects

5.1.1 View Management

The View is responsible for registering itself with its Displayer after it has been constructed. Registration is necessary because the Displayer needs to know which Views it is responsible for and must assign a menubar to each of its Views. When the View registers itself with its Displayer, it obtains a unique menubar identifier which it can use to build up its menus. Once this is complete, it can call the *setView* method in the Displayer to insert itself into the Displayer. The Displayer is responsible for keeping track of which Views have been inserted and which have merely been registered.

When a View removes itself from the Displayer, it is not de-registered. That is, the Displayer still knows about the presence of that view as it may wish to be re-inserted in the window. De-registration destroys the View and its menubar, and if that particular View is required again by the application it must be reconstructed. If a View has not been removed from the Displayer before de-registration, it will be removed before destruction. The focus passes to the next View in the list of inserted Views.

5.1.2 Menu Management

When the View registers itself with the Displayer, it is assigned a unique menubar identifier based on a unique label provided by the View. The Displayer creates this menubar for use by the View. After this, the View can operate on the menubar independently of the Displayer. The View can indicate to the Displayer that it would like it to display its menubar by configuring the menubar option. A View can also query its menubar identifier by using the *menubar* method.

Adding and removing menus and menu items is accomplished by calling the appropriate Menubar methods which are shown in Figure 8. Interface details of the Menubar object are described in the final part of this report.

5.2 Views

Views are responsible for rendering design information. Each type of View knows how to display specific types of information. For example, a File View knows how to display files and is responsible for providing a mechanism to perform operations on a file such as opening, closing, and editing. A View may employ a Data Filter to either convert a particular representation of data that must be rendered into a form that it will understand or to filter out unnecessary pieces of information that a user may not want to see.

It can also incorporate sub-views which allow different forms of design information to be combined in a single View. If sub-views are present in the View then the View's menubar will contain a superset of all the menu items required for all views. A sub-view is not allowed to manipulate the parent View's menubar. The parent is responsible for creating its menubar. The View methods and attributes are shown in Figure 8 and the implementation details are discussed in Section 6.4.

5.3 Data Filters

A Data Filter (DF) transforms one data representation to another. It may filter out pieces of information that may not be required in the new representation. Data Filters may be cascaded as shown in Figure 9 allowing data to go through multiple transformations before being passed on to a View for rendering. One of the advantages of employing Data Filters is that different views of the same basic information can be easily generated by interchanging one filter with another.

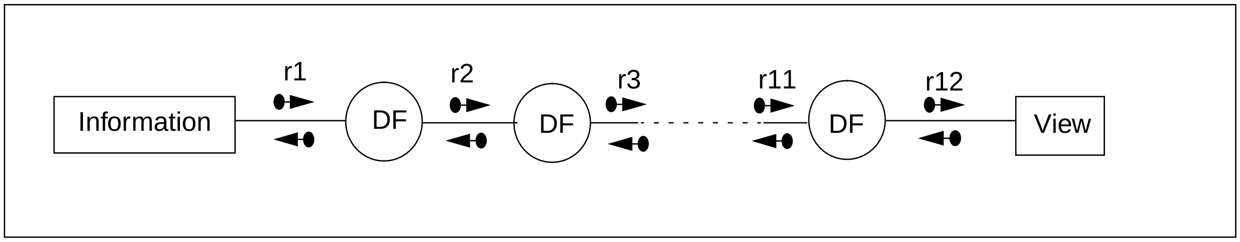


Figure 9. The Data Filter Concept

Each Data Filter knows the relationship between the original piece of data and the transformed counterpart hence it can save an edited version of the new representation in the pre-processed form. Each Data Filter must keep track of any filter preceding it as it would need to communicate with its predecessor when it passes back edited information which may need to be saved in its original form. The concept of a Data Filter is fairly general so each particular type of Data Filter will have differing details. The attributes and methods common to all Data Filters are shown in the figure below.

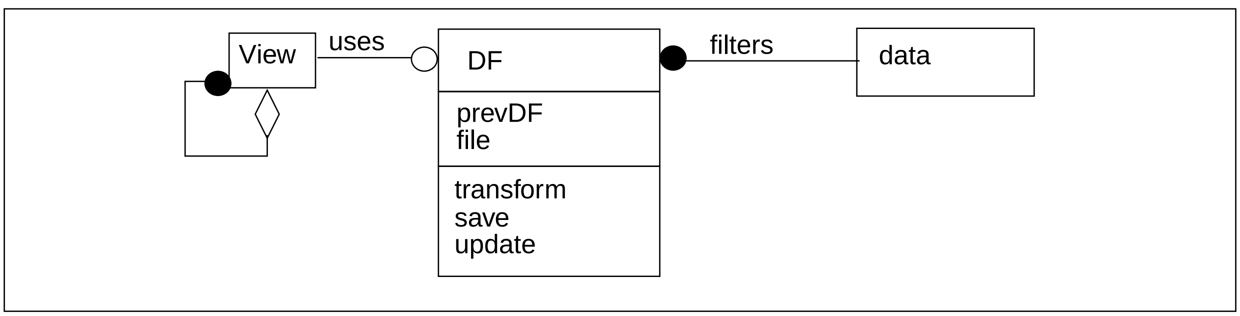


Figure 10. Data Filter

5.4 Applications

The Displayer-View-Data Filter paradigm presented above can be useful for displaying many different types of design information such as schedules, performance metrics, architecture diagrams, and class browsers. This project concentrates on applying this framework to displaying visual representations of algorithm and architecture interaction. This is discussed in the next section.

6 Implementation of the Object-Oriented Model For Design Visualization

This section describes the final part of this project – the implementation of the object-oriented model for design visualization and the implementation of the techniques used to visualize algorithm and architecture interaction in embedded systems. The prototype is written in [incr Tcl] and [incr Tk] [14], and is part of the Tycho project. As mentioned in the previous section, the two key classes are the Displayer and the View. The functionality of each was described in detail earlier. The Data Filter manipulates design information or data and presents it to another Data Filter or a View in a different form. Data Filters can be cascaded to allow for editing interrelated data or enabling multiple views of the same data. An overview of the interaction between these three main classes is shown in Figure 12. A summary of the graphical notation is provided in the figure below.

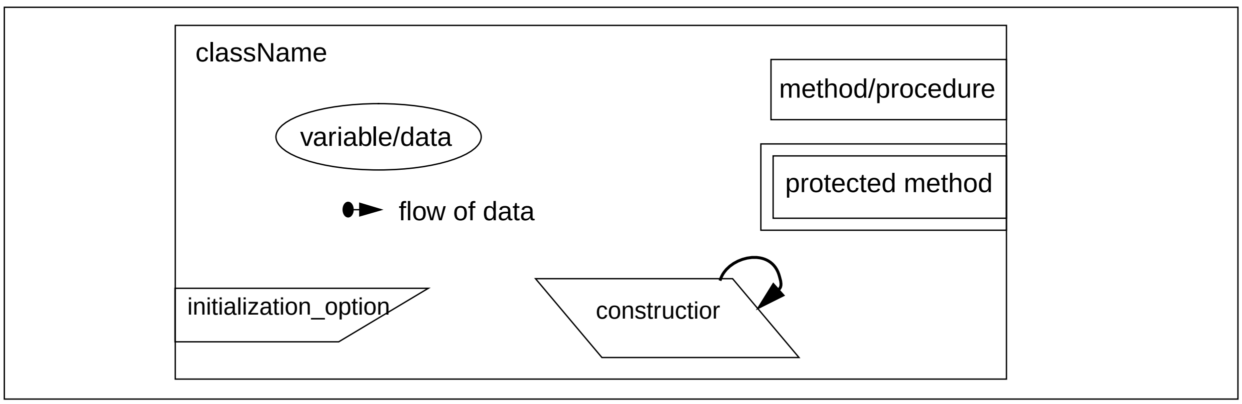


Figure 11. Summary of Notation Used to Depict Implementation

6.1 Overview

Tycho, the syntax manager for Ptolemy, is written entirely in [incr Tcl] and [incr Tk]. The Tycho kernel is comprised of a number of base classes that are used in almost all editors and displayers. However, the *slate,* which provides a basic drawing surface for graphical objects and pictures, serves as a base for the Tycho visual language toolkit and is independent of the kernel. The classes developed in this project use the *slate* (and other associated classes) as well as some of the classes in the Tycho kernel.

The Displayer and View interact in the following manner. When a View wants to use a Displayer, it will register itself with the Displayer by providing the *registerView* method with a label and its identifier. The *registerView* method acknowledges the registration by returning a unique menubar identifier which the View can then use to create its menus and menu items. Once the View has completed its setup, it asks to be displayed by calling the *setView* method. In response, the Displayer inserts the appropriate menubar and the View grabs the focus. A View can remove itself from the Display simply by calling the *removeView* method. Even though a View may not be displayed, it may still be registered with the Displayer. To deregister itself, the View must call the *deregisterView* method which will destroy the View and its menubar. After having called the *deregisterView* method, an application desiring to re-display the View must reconstruct and register it again with the Displayer.

The View can employ the services of a Data Filter by specifying a particular instance of a specific Data Filter as an initialization option. The View is responsible for calling the *create* method of the Data Filter to create any appropriate data structures that it may require to transform data from one representation to another. Each specific type of Data Filter may be very different from another; hence, the abstract Data Filter class is rather sparse.

Details of each class are described in the following subsections followed by a description of how the techniques used to visualize algorithm and architecture are implemented.

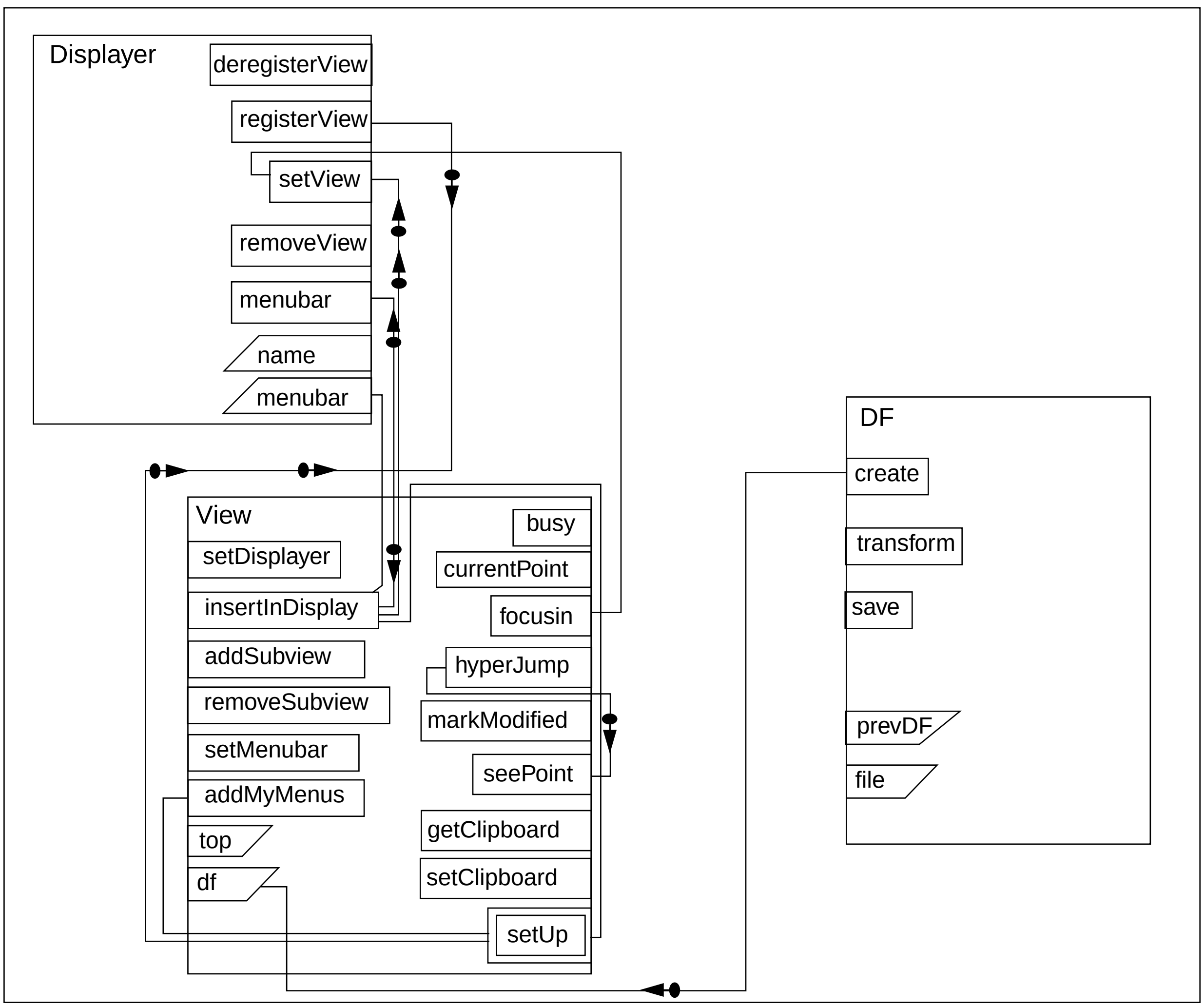


Figure 12. An Overview of the Implementation of the Model for Design Visualization

6.2 Displayer

The Displayer inherits from the Tycho *Dismiss* class which provides basic window functionality and a status bar. It has two options that can be configured to give the Displayer a name and a menubar. Methods exist to insert and remove Views. Upon construction, the Displayer will set up frames to house a menubar and views. It will also create a default menubar that is visible when there are no views in the Displayer. The figure below describes the implementation in graphical form. Details regarding the methods and variables follow.

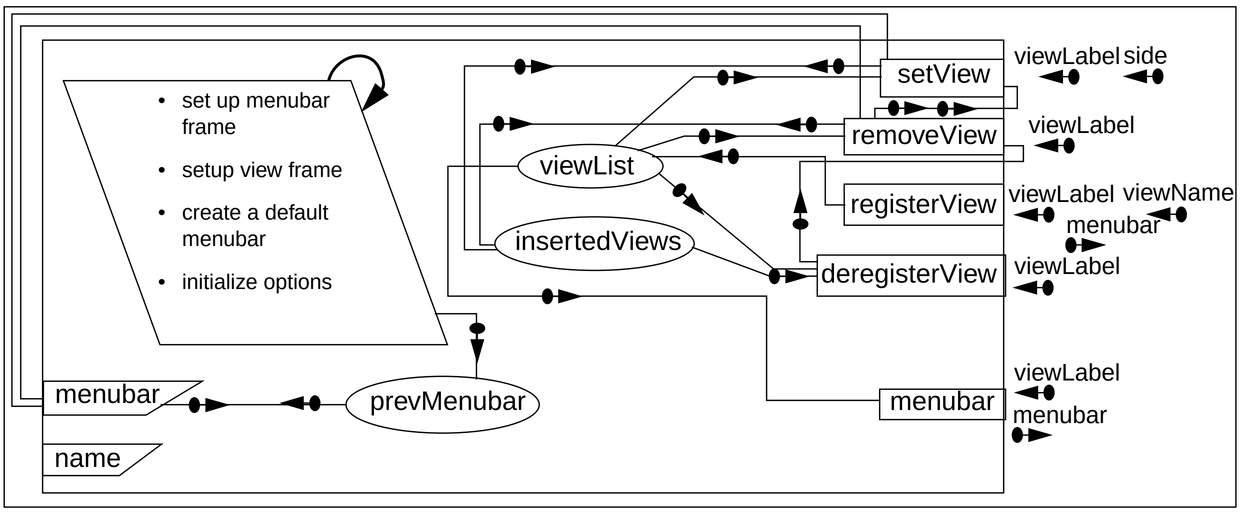


Figure 13. Implementation of Displayer

6.2.1 Displayer Options

The Displayer has two options: *menubar* and *name*. The *menubar* option allows a user to configure the Displayer with a user-specified menubar. The *name* option gives the Displayer a name which is displayed as a window title.

6.2.2 Displayer Methods and Variables

The Displayer has four methods that manage insertion and removal of views. The fifth method provides a mechanism for a View to query its menubar identifier. The *menubar* method requires the View to provide it with a label which is used to obtain the unique menubar identifier. If the label argument is an empty string then the method will return the name of the default menubar. The *registerView* method allows a View to register itself with the Displayer. It requires the View to provide it with its name and a unique label. The label serves two functions – it is used as an index into the list of registered Views (*viewList*) and inserted Views (*insertedViews*), and to create a unique menubar identifier for each View. Once a View is registered with the Displayer, it is inserted into the *viewList* array which keeps track of all registered Views. Views cannot be inserted into the Displayer unless they have been registered. A View can deregister itself by calling the *deregisterView* method which uses the label parameter to find the View in the *viewList* and then removes it from the list. The View and its menubar are consequently destroyed.

If a View has been inserted into the Displayer and *deregisterView* is called, then it is first removed from the Displayer by calling the *removeView* method. The *removeView* method does not destroy the View and its menubar; it only removes it from the list of inserted Views. If the removed View had the focus, then its menubar is removed as well and focus is given to the next View in the list of inserted Views. A View can be inserted into the Displayer by calling the *setView* method which will place the View either at the top or bottom of the Displayer or to the left or right of previously inserted Views. The *side* argument to the method provides this functionality. The *prevMenubar* variable keeps a record of which menubar was being displayed before the menubar option changed. The next section describes the Menubar widget which is associated with each view.

6.3 Menubar

The Menubar widget may be inserted into any Displayer or top level window. It provides methods to add and remove menus and menu items, and to insert menu separators. It also allows a user to enable and disable menu items and entire menus. Each View is given a menubar when it registers itself with a Displayer. The View can then operate on its menubar independently of the Displayer. Figure 14 shows the interface.

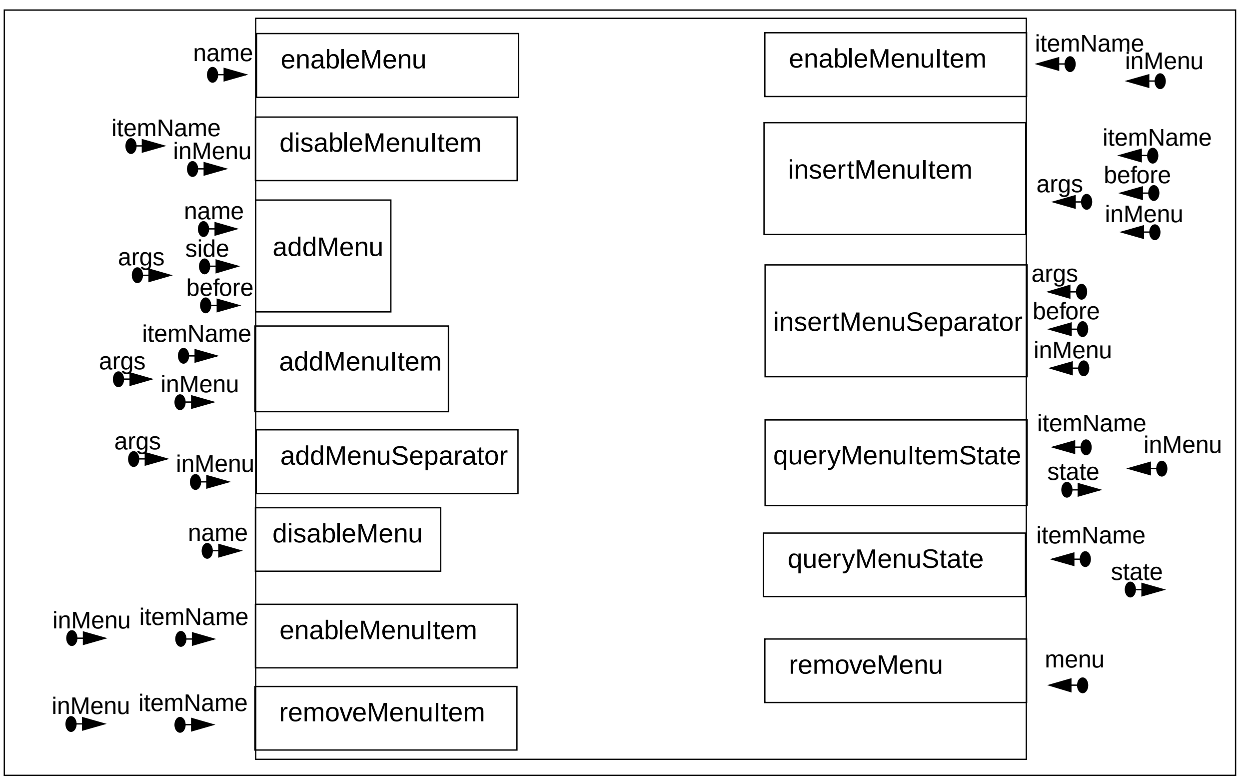


Figure 14. Implementation of Menubar widget

6.3.1 Menubar Methods

The menubar widget provides a simple way of creating menubars. Adding menus is easily accomplished by calling the *addMenu* method and adding menu items is done by calling the *addMenuItem* method. Removing menus and menu items is done in a similar fashion by calling the respective remove methods. Similarly, menus and menu items can be enabled and disabled using the enable and disable methods. Menu item separators can be added using the *insertMenuSeparator* and *addMenuSeparator* methods.

6.4 View

The View is an abstract class that other Views which specialize in rendering certain types of information can inherit from. The abstract View class provides guidelines on how Views should communicate with the Displayer. It also provides some generic methods required by all editors and displays. The implementation is shown in graphical form below. Following the figure, a description of the options, variables, and methods is given.

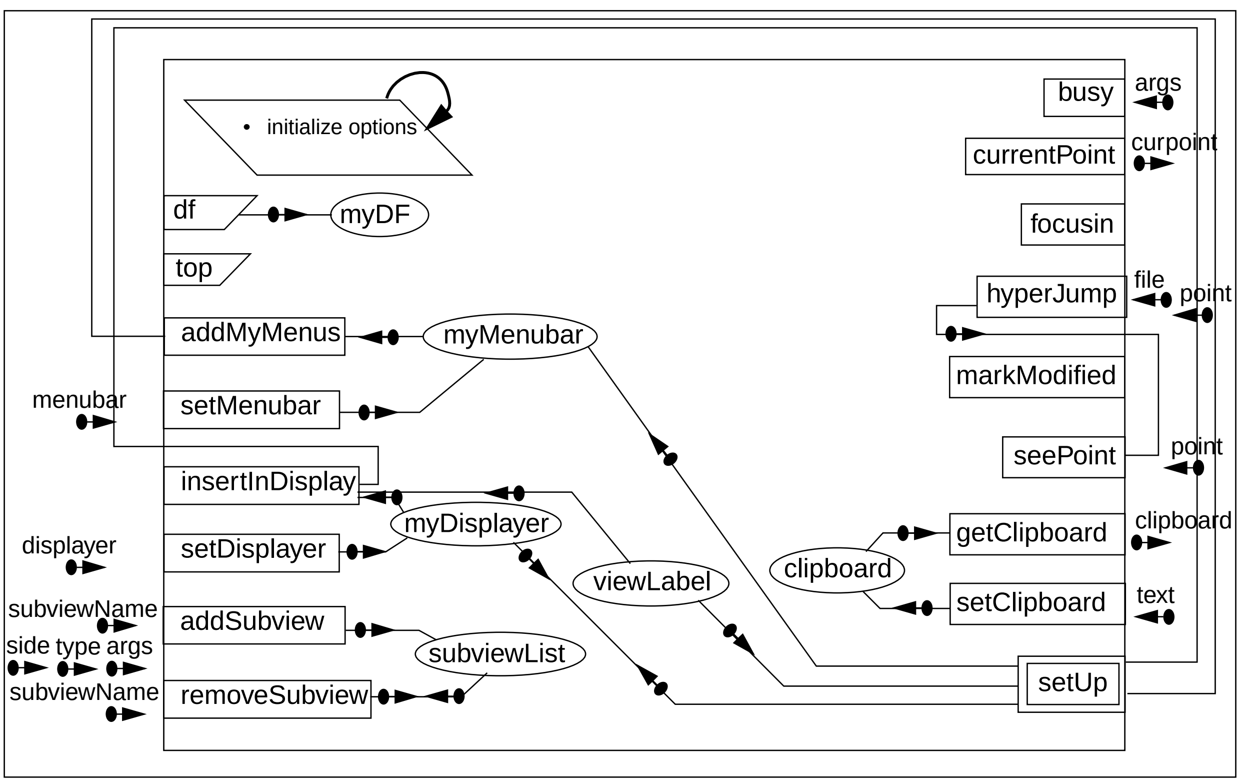


Figure 15. Implementation of View

6.4.1 View Options

The abstract View class has two configurable options. The first, *df*, specifies the name of the Data Filter that a View may want to employ to filter some data into a form that it can easily understand and render. The second option, *top*, allows the user to specify the top level window that contains the View.

6.4.2 View Methods and Variables

The first View method, *addMyMenus*, does not do anything, however, derived classes should redefine this method to perform all operations that are required to create its menubar. This method is called from the protected method *setup*. The *setup* method sets the Displayer for the View, registers the View with the Displayer, and creates the unique label required for registration purposes. The final task in setup is to create the menubar for the View by calling the *addMyMenus* method. The *insertInDisplay* calls the setup method and then calls the Displayer method *setView* which inserts the View into the Displayer. The newly inserted View grabs the focus. The names of the Data Filter, Displayer, and Menubar are saved in the variables *myDF*, *myDisplayer*, and *myMenubar*. The unique label for the View is preserved in *viewLabel*.

Views may have subviews. This allows for mixing of different types of syntaxes (e.g graphics and text). The *addSubview* and *removeSubview* methods provide mechanisms to add and remove subviews. A list of all subviews within a single View is contained in the variable *subviewList*. The methods *setMenubar* and *setDisplayer* provide a way to change the View's Displayer and Menubar. At the present, subview functionality has not been implemented due to some issues that have not yet been resolved. For example, there is the issue of how to set up the menubar when the View contains many different subviews. A lot of effort needs to be spent on designing the View so that it can cleanly support subviews.

Other methods associated with the View are general and are required for all editors and displays. These methods were originally written for a standard Tycho widget by Edward A. Lee but now have been integrated into the View. The reader is asked to look at Tycho documentation for their details.

6.5 Slate View

The Slate View was developed to act as a scrollable drawing surface that could support graphical objects and pictures. John Reekie, a Tycho developer, has created a *slate* which was designed for this purpose but does not support scrollbars or menus. The Slate View inherits from View and embeds a slate to which it adds scrolling capabilities. A lot of the code required to implement scrolling was borrowed from the code written for the scrollable canvas widget created by Sue Yockey and Mark Ulferts. The Slate View is used as a base class for all the graphical Views developed for the tool that visualizes architecture and algorithm interaction. A graphical description of the implementation is not given as Slate View only adds extra methods and options to those already existing for View. The reader should refer to Tycho documentation for more details regarding the *slate*.

6.5.1 Slate View Options

The *automargin* option gives the size of the margin between the edges of the slate and the bounding box containing all the items on the slate. The *autoresize* option, when set, allows the scrollbars to adjust automatically when new items are added to the slate. The height and width of the slate are given by the *height* and *width* options. Scrollbars can be dynamic or static – that is, they can be set so that they disappear when they are no longer required. The *hscrollmode* and *vscrollmode* options specify this for the horizontal and vertical scrollbars, respectively.

6.5.2 Slate View Methods

The Slate View contains all the methods required by the View and in addition provides methods for scrolling the slate. The *childsite* method returns the name of the slate that is embedded in the view. Also, wrappers have been created for all the slate methods so that the same operations can be performed on the scrollable slate. For the sake of brevity, details will not be provided here as Tycho documentation already contains this information.

6.6 Data Filter

Data Filters that operate on specific types of data inherit from the Data Filter abstract base class. It provides some general methods to operate on data, however, derived classes are expected to overload the methods with their own definitions as Data Filters can be very different from one another. The abstract class implementation is shown below in Figure 16. Following it is a description of its options, methods and variables.

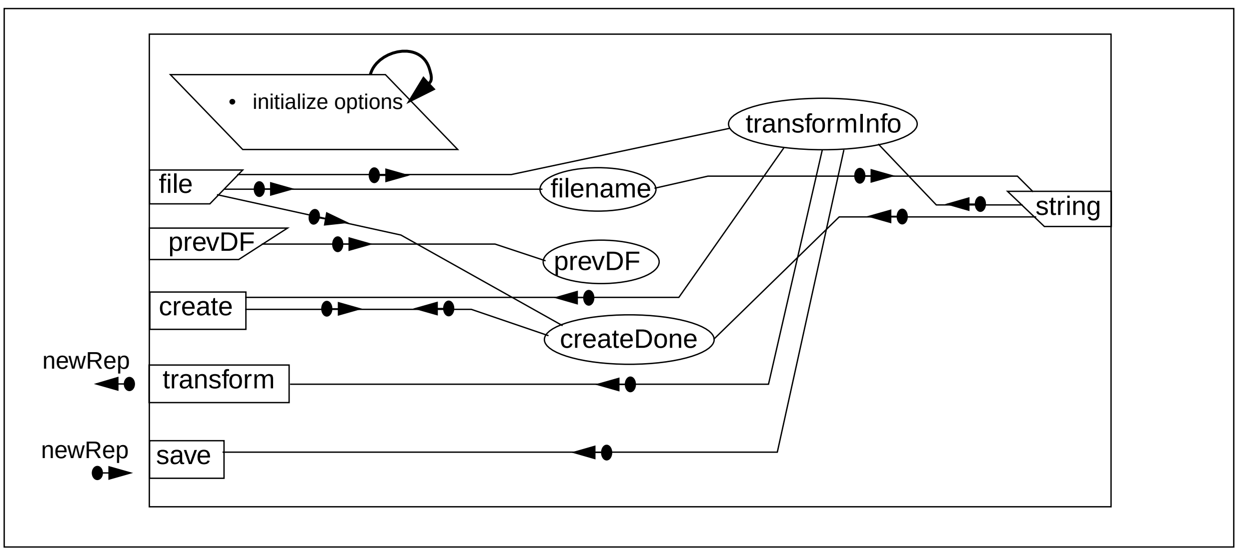


Figure 16. Implementation of a Data Filter

6.6.1 Data Filter Options

The Data Filter has three options: *file*, *prevDF*, and *string*. The *file* option is used to specify the name of the file that will constitute the input to the Data Filter. A string may be specified instead of a file; this is done by configuring the *string* option. If both options happen to be configured, preference is given to the *file* option. The *prevDF* option specifies a "previous" Data Filter that processes data before it is passed to the "current" Data Filter. This option allows cascading of multiple Data Filters.

6.6.2 Data Filter Methods

There are three methods associated with the current implementation of the Data Filter. The *create* method creates any data structures within the filter that may be required when data is transformed from one representation to another. The *transform* method performs this transformation. The *save* method saves a modified version of the new representation in the form of the old representation. In other words, it tries to perform the transformation operation in reverse. It should be noted here that the code for the reverse transformation has not yet been written. More effort is still required in this area as reversing a filtering operation is normally very difficult.

6.7 Architecture View

The Architecture View represents the visualization of embedded system hardware components. This section provides an overview of a possible implementation. It should be noted that at the present, the Ptolemy target object does not have the capability of writing out an architecture description to a file. Hence, no code has been written for this view.

In order to help a visualization tool render a graphical display of system components, it is necessary to provide the tool with some sort of representation of the architecture that can easily be parsed by the tool. A file similar to the schedule file described in Section 3.3.2 is used to perform this function. The format is given in Table 8.

Table 8. *Architecture File Format*

Entry	Format
<*target*>	{ (<*entry*>)+}
<*entry*>	{ **target** <string:target_identifier> } | { **block** <string:block_identifier> <*target*> } | { **connect** <string:toBlock> } | { **connect** <string:toBlock> <*connection_info*> } | { (<*estimate*>)+ } | { (<*num*>)+ }
<*connection_info*>	{ **connected** } | { **not_connected** }
<*estimate*>	{ **send** <int: exec_time}>} | { **receive** <int: exec_time> } | { **utilization** <float: utilization> } | { **throughput** <float: throughput> } | { **avgMemAccess** <float: memAccess> }
<*num*>	{ **numberOfBlocks** <int:numblocks> } | {**numberOfInputs** <int:numinputs> } | {**numberOfOutputs** <int:numoutputs> }

A Data Filter similar to the one described for the Gantt View may be developed to parse the architecture file and generate the display data.

6.8 Gantt View

The Gantt View class is responsible for creating a graphical representation of a schedule. It uses the Gantt Data Filter, GanttDF, to query important information from the schedule file. Details regarding the Gantt Data Filter are given in a later section. Below is a figure which presents a graphical view of the implementation. Following it is a description of the options and methods.

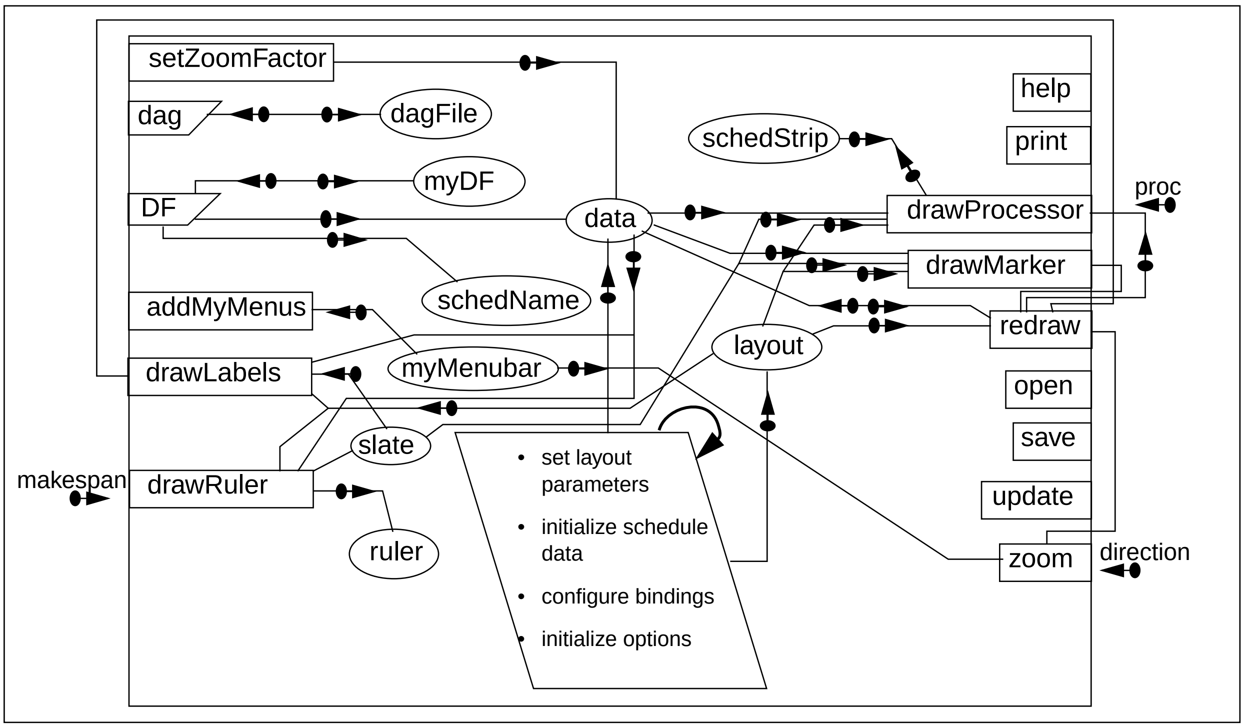


Figure 17. Implementation of Gantt View

6.8.1 Gantt View Options

The Gantt View has one option in addition to the options present in its super-class, View. The *dag* option specifies a Directed Acyclic Graph (DAG) representation of the precedence constraints in a data flow graph which can be used to validate any editing operations that may be made on the schedule displayed by the Gantt View. Since editing capabilities have not been implemented, the *dag* option does not perform any useful function at this time. It is hoped that in addition to the *dag* option, there may be other options that would allow easy validation of a modified schedule – for example, an annotated SDF representation.

6.8.2 Gantt View Methods and Variables

The *addMyMenus* method adds the appropriate menus and menu items to the menubar supplied by the Displayer. Menus include File, Edit, Zoom, Legend, and Help. The File menu contains items pertaining to operations normally associated with files; the Edit menu contains items that allow a user to edit a Gantt Chart; the Legend provides information regarding the meaning of colours, lines, and shapes. The Help menu provides the user with information pertaining to the use of the Gantt Chart.

A number of methods exist that help with drawing various parts of the Chart. The *drawLabels* method is responsible for drawing processor labels, and schedule identifiers in the appropriate places. The *drawMarker* method creates a bright marker that is positioned wherever the user clicks the mouse. The marker is useful for finding out what tasks are executing at different time instances. The position of the marker with respect to the time axis is displayed above the marker. The *drawProcessor* method creates a graphical representation of the schedule on the *slate*. A *strip* of blocks or other objects such as circles are associated with each processor. The *drawProcessor* creates the strip from information provided by the Gantt Data Filter. This filtering operation is discussed in a later section. The *drawRuler* method draws the time axis that appears above the schedule.

Other methods are associated with the menu items. These include *open*, *close*, *print*, *zoom*, *setZoomFactor*, and *help*. The names of the methods indicate their function. The *redraw* method is called each time the View is resized and it redraws the entire chart so that it fits inside the Displayer.

6.8.3 Gantt Data Filter

The Gantt Data Filter parses a schedule file and provides methods that a Gantt View can use to query important pieces of information that are required to draw a graphical representation of the schedule. For each schedule file, the filter searches for nested schedules and when it finds one (indicated by key words such as **cluster**, **preamble**, **processor, repeat**), it creates a child Gantt Data Filter. This recursive operation results in a tree of Data Filters that process different parts of the schedule file. Each individual Data Filter parses the schedule that it is responsible for and generates a string that has the following form: {*type*, *proc*, {*name*, *start*, *end*}+}. The *type* indicates what type of graphics the Gantt View should use to render the information. As shown in Table 7 different schedules have different graphical representations so the View should be able to delineate between the various types. The *proc* entry tells the View which processor is associated with the given schedule. The list of block names, start times, and end times is used to draw the individual tasks in whichever form is appropriate for that schedule.

6.8.4 Modifying Schedules

It is not yet possible to modify schedules using the Gantt View. There are a number of issues that must be addressed before proceeding with the implementation. These were outlined in Section 4.3.1. A possible implementation strategy is outlined below for future reference.

- A data structure that represents a Gantt Chart is required. This would make it easier for saving Gantt Charts and changes made to them.
- A mechanism to specify constraints on editing needs to be developed. One could envision somehow attaching a constraint to the editor or an editing operation. If a change is not allowed then it simply will not be performed and the user will be notified. If the change affects other parts of the schedule then these changes should be automatically reflected in the Gantt Chart.

- With regards to validation strategies, there are two possible routes: use a compact version of the APEG to validate the schedule, or use an annotated version of the SDF graph. The simplest strategy is to simulate the change and check whether deadlock occurs. It may be useful to give a user the choice of using whichever strategy they prefer.

6.9 Trace View

As mentioned earlier, Ptolemy can generate execution traces, however, at the present time the traces are not generated in a format that is useful to a visualization tool. Table 9 describes a format more amenable to visualization. The trace can consist of estimated and actual execution behaviour.

Table 9. *Trace File Format*

Entry	**Format**
<*trace*>	{ (<*entry*>)+ }
<*entry*>	{ **actual** <*f_info*> } | { **estimate** <*f_info*> }
<*f_info*>	{ <*star*> { **exec_time** <int: exec_time> } }
<*star*>	**send** | **receive** | <string:star_name>

The simplicity of the file format makes the use of a Data Filter unnecessary. This View has not yet been implemented.

6.10 Communication View

The Communication View shows how processing units or resources communicate with one another over time. The schedule file contains communication information however, it does not indicate from where processing units are receiving information and to where they are sending it. A modification is necessary. The receive and send entries should be augmented with source and destination fields. Once this is done it will be a simple task to filter this information from the output of the Gantt Data Filter. The Communication Filter need only search for the key words **send** and **receive** in the Gantt Filter output and transmit those items to the Communication View for rendering. The figure below shows this concept.

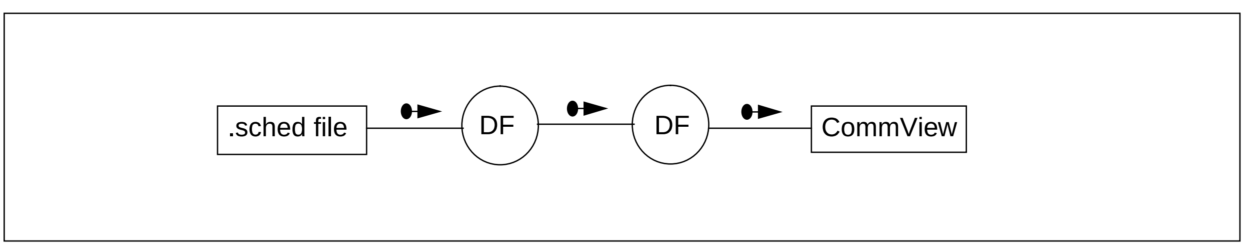


Figure 18. Generating Communication Pattern View from a Schedule File

7 Summary and Future Work

Embedded systems are quickly becoming integral parts of our daily lives. They appear in consumer electronics, automobile parts, medical technologies, and telecommunications equipment. The complexity associated with such systems makes it very difficult for engineers to exploit the full potential of the underlying system resources. This often results in sub-optimal performance. Tools that allow an engineer to easily assess system behaviour and performance can reduce development costs and time-to-market.

In the past, visualization techniques have proven invaluable to the design process as they have simplified tasks faced by engineers. Techniques that can highlight poor algorithm design, problematic hardware-software interfaces and other reasons behind poor performance can greatly simplify the embedded system design process. The work presented in this report shows that visualization of algorithm and architecture interaction is an important aspect of performance-based design. Four views of algorithm and architecture interaction have been developed based on visualization fundamentals and Gantt Charts. An object-oriented framework for editing and displaying various types of design information has been designed, described, and implemented. The work on implementing the visualization of algorithm and architecture interaction within this framework has been initiated as well.

A lot of effort still remains to be spent in implementing the remaining parts of the tool that will visualize algorithm and architecture interaction. However, the implementation of the object-oriented framework for displaying design information is complete. Editors and displayers currently available in Tycho will move over to the Displayer-View-Data Filter paradigm in the very near future. Work on implementing functionality to back annotate schedules is still required and constitutes a large project in itself. The Data Filter concept has shown promise but more detailed work needs to be done regarding its development as it is still a fairly new idea. In conclusion, it would be interesting to explore better visualization techniques and apply them to embedded system design.

8 References

[1] M. Abrams, N. Doraswamy, and A. Mathur, "Chitra: Visual Analysis of Parallel and Distributed Programs in the Time, Event, and Frequency Domains", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 6, November 1992, pp.672 - 685.

[2] M. A. Bayoumi (ed.), *VLSI Design Methodologies For Digital Signal Processing Architectures*, Kluwer Academic Publishers, USA, 1994.

[3] S. S. Battacharyya, P. K. Murthy, E. A. Lee, *Software Synthesis From Dataflow Graphs*, Kluwer Academic Publishers, USA, 1996.

[4] J. P. Calvez, *Embedded Real-Time Systems: A Specification and Design Methodology,* John Wiley & Sons, West Sussex, England 1993.

[5] J. B. Dennis, "First Version Data Flow Procedure Language", Technical Memo MAC TM61, May, 1975, MIT Laboratory for Computer Science.

[6] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, PTR Prentice Hall, Englewood Cliffs, N.J., USA, 1994.

[7] H. L. Gantt, "Organizing for Work", *Industrial Management*, August, 1919, pp.89 - 93.

[8] M. T. Heath, J. A. Etheridge, "Visualizing the Performance of Parallel Programs", *IEEE Software*, September 1991 pp. 29 - 39.

[9] M. T. Heath, A. D. Malony, and D. T. Rover, "Parallel Performance Visualization: From Practice to Theory", *IEEE Parallel and Distributed Technology*, Winter 1995, pp. 44 - 60.

[10] M. T. Heath, A. D. Malony, and D. T. Rover, "The Visual Display of Parallel Performance Data", *IEEE Computer*, November 1995, pp. 21 - 28.

[11] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, USA, 1990.

[12] D. D. Hils, "Visual Languages and Computing Survey: Data Flow Visual Programming Languages", *Journal of Visual Languages and Computing*, Vol. 3, 1992, pp.69-101.

[13] Hypertext Webster Interface. http://gs213.sp.cs.cmu.edu/prog/webster?

[14] [incr Tcl] and [incr Tk]. http://www.tcltk.com/itcl/index.html

[15] M.-H. Jobin, P. Lefrancois, B. Montreuil, "Using a Public 3-D Gantt Chart Communication Structure between Agents for Distributed Scheduling Architectures", *Integrated Computer-Aided Engineering*, Vol. 1, No. 2, pp. 147 - 156, 1993.

[16] G. M. Karam, "Visualization using Timelines", *1994 International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, WA, USA, August 17 – 19, 1994.

[17] J. Kodosky, J. MacCrisken, and G. Rymar, "Visual Programming Using Structured Data Flow", *Proceedings of IEEE Workshop on Visual Languages*, October 1991, Kobe Japan.

[18] E. A. Lee, A. Kalavade, W.-T. Chang, "Effective Heterogeneous Design and Co-simulation", *NATO Advanced Study Institute Workshop on Hardware/Software Codesign*, Lake Como, Italy, June 18 – 30, 1995.

[19] A. D. Malony, D. H. Hammerslag, and D. J. Jablonowski, "Traceview: A Trace Visualization Tool", *IEEE Software*, September 1991 pp. 19 - 28.

[20] P. Osmon and P. Sleat, “IDRIS: Interactive Design of Reactive Information Systems”, *Proceedings of Fourth International Conference on Advanced Information Systems Engineering*, Springer-Verlag, Germany, 1992, pp. 494 - 506.

[21] E. K. Pauer and J. B. Prime, “An Architecture Trade Capability Using the Ptolemy Kernel”, *1996 International Conference on Acousitics, Speech, and Signal Processing*, Atlanta, Georgia, May 7 - 10, 1996.

[22] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, “A Hierarchical Multiprocessor Scheduling System for DSP Applications”, *Twenty-Ninth Annual Asilomar Conference on Signals, Systems, and Computers*, October, 1995.

[23] J. L. Pino, S. Ha, E. A. Lee and J. T. Buck, “Software Synthesis for DSP Using Ptolemy”, *Journal of VLSI Signal Processing*, 9, 7-12, 1995.

[24] Ptolemy Team, *The Almagest, Vol.1-4, Ptolemy 0.5 Kernel Manual,* EECS, University of California, Berkeley, 1990-1994.

[25] J. Rambaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, Inc., Englewood Cliffs, N.J., USA, 1991.

[26] G. C. Sih, *Multiprocessor Scheduling to Account For Interprocessor Communication*, Ph.D. Dissertation UCB/ERL M91/29, University of California, Berkeley, 1991.

[27] C. M. Woodside, “A Three-View Model for Performance Engineering of Concurrent Software”, *IEEE Transactions on Software Engineering*, Sept. 1995, vol.21, No.9, pp. 754 - 767.

[28] J. Zhu, et. al., “HaRTS: Performance-Based Design of Distributed Hard Real-Time Software”, *Journal of Systems and Software*, February, 1996, pp. 143 - 56.