The Integration of Control and Dataflow Structures in Distributed Hard Real-Time Systems

C. Puchol* and A. K. Mok†
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188

Abstract

The control structure of many real-time applications are naturally described by state machines. However, the state transitions of these machines are governed not only by the state of the system and the occurrence of events, but also by their time of occurrence. The control structure of a real-time system determines what computation to perform and the set of timing constraints in effect at all times. The computation performed by the system can then be modeled by a directed graph, where the nodes denote transformations on the data and the edges denote data flow. In this paper, we discuss some research issues that arise from the integration of control flow with data flow.

1 Introduction

A primary objective of the SARTOR project at the University of Texas at Austin is the mechanization of the design chores in building distributed hard-real-time systems. Full automation is difficult if not impossible for the engineering of large general-purpose computing systems. It is therefore important to carefully restrict oneself to a proper domain model. In [4] we have summarized our domain model for real-time systems by means of an "equation" which is meant to capture the structure of a hard-real-time system:

Real-Time System =
State Machine + Timing Assertions + Dataflow Model

Much work has been done by real-time system researchers in advancing the analysis of state machines ([2], [3]) and schedulability determination of data transformation tasks ([1]). However, there has been little done in examining the integration of the control structure (state machines) with the dataflow structure (data transformation) for real-time systems. In particular, timing assertions are associated with the throughput and delay bound requirements on data flow in a system but timing constraints are derived from the behavior of the state machine. On the other hand, behavior of the state machine is determined by the outcome of data transformation computation.

We present the structure of a distributed real-time system in Section 2, followed by a real-world example. Section 4 focuses on the structuring issues presented in the system implementation. Section 5 defines what are the data-flow models on which the discussion is based.

2 The Structure of a Distributed Hard Real-Time System

In traditional real-time applications, the role of the state machine is to determine the "mode" in which a system is operating. A change in the mode quite often requires the modification of the timing assertions on the computation elements (e.g. sampling rate for a data acquisition module) as well as the number of active computations. In general, the state machine initiates mode changes and determines what parameters need to be changed in which computation elements, as well as the tasks to be added or deleted to the currently executing task set (and their corresponding timing constraints).

In an implementation, mode changes must be effected by *signals*. The state machine and the computational elements in the dataflow graph communicate by passing *signals*. A naive abstraction is for each one of the functional modules in the dataflow graph to have an associated state machine (presumably small) which will receive the signals intended for the module and will modify its behavior accordingly. It will also send signals to the state machines of other functional modules appropriately. We refer to this state machine as the associated state machine of the functional modules.

This poses some interesting questions in the decomposition problem as discussed below. After the specification of the system is provided, including the timing assertions for the control and dataflow structures, the first step towards implementation is to break up control and dataflow structures so that each subset of these structures can be assigned to separate processing elements. We call this the *decomposition* phase. The result of a decomposition produces a set of *schedulable units* which, along with specific timing behavior information will determine the performance obligation of the run-time system.

The naive abstraction of the control structure assumes that the global state machine has been or can be conveniently decomposed into individual state machines whose product yields the same behavior. In general, this is far from assured. In the case when this is possible, the state machine encapsulating the control structure of the system can be implemented as a single task with its own timing constraints. These control tasks will most likely have tight timing constraints since they must track the progress of the

*Work sponsored by a Fulbright Scholarship
†Work sponsored by Office of Naval Research under contract no. N00014-89-J-1472

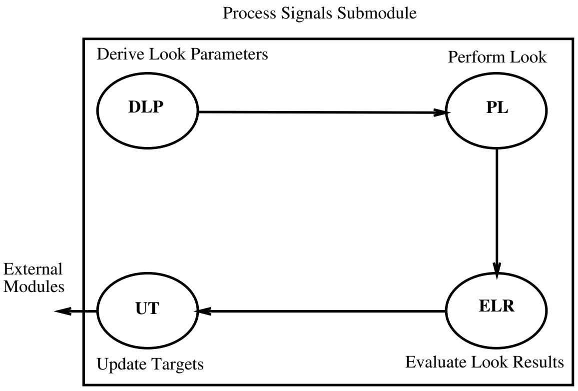

Figure 1: Dataflow graph of the Process Signals submodule in the HLCS missile.

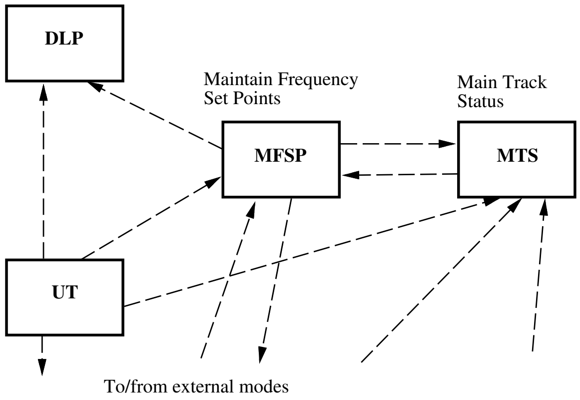

Figure 2: Part of the HLCS control structure, associated with the Process Signals submodule.

system at all times. A common assumption is that since it is essentially a state machine, there ought not be any unusually heavy computation associated with it. However, the conditions that trigger transitions are often caused by the values of data resulting from costly computation (e.g., cpu-intensive signal processing algorithms), and hence even though (local) state transitions are cheap to implement, the delay between transitions cannot be ignored.

3 A Real-world Example

To make our discussion more concrete, we give an example of the integration of the control structure and the dataflow structure of a model of the HLCS missile system[1].

In this example, a submodule called "Process Signals" scans the external environment of the missile and processes all the signals while analyzing and keeping track of them. The information is then passed to some other external processes. The environment of the system is sampled in terms of *looks* which is the unit used in the Process Signals module. A look is a full sample of all the antennas and radar data gathered from the missile environment.

The functional modules in the dataflow graph of the system are defined as follows (See Figure 1):

- **Derive Look Parameters:** this module provides all the parameters for the sampling of the look.
- **Perform Look:** initiates the sampling devices for the data gathering and collects the data which is later sent for evaluation.
- **Evaluate Look Results:** this module processes the signals in the look and identifies the signals obtaining concrete data about the targets in the missile environment.
- **Update Results:** updates the track database for each of the targets in the look.

The functional modules in the dataflow graph are executed in pipeline periodically. More than one look may be in the system at a time. The periodicity of the sampling is dictated by the control structure of the system. In certain occasions, the mode of the system may change depending on the data sampled from the environment. For instance the system may be tracking several targets, when at some point, the control structure determines the need to track only one target and at that it should be done at a higher rate. The parameters in the sampling of the look are drastically changed. The block diagram of the control structure associated with the Process Signals submodule is showed in Figure 2. Each of the control modules is a timed state machine (Statechart, Modechart, etc.).

The control module **MFSP** maintains frequency set points for the different radiation antennas and detectors. It receives signals from UT and MTS as well as from the outside. The **MTS** module maintains what is the status of the tracks of the targets currently being traced. It communicates with MFSP and UT.

[1] HLCS: HARM Low-Cost Seeker. HARM: High speed Anti-Radiation Missile System.

4 Structuring Issues

4.1 Synchronous vs. Asynchronous Implementation of State Machines

As was mentioned in a previous section, there may not be any viable way to decompose the global control structure into individual state machines associated with each functional module. In the application example, note that there are two modules (DLP and UT) which are common in both structures. Thus in this example, some but not all of the functional modules in the dataflow graph may have a control structure associated with them. The reason is that in an arbitrary decomposition, signals might have to propagate with sufficiently long end-to-end delays (including delays introduced by data transformations which cause signals to be emitted) such that the individual state machines can no longer be regarded as acting in synchrony (i.e., all machines perform local transitions and will not signal other machines until **all** machines become quiescent). While software implementation techniques are well developed for synchronous machines, software implementation of asynchronous machines are another matter.

4.2 Signal vs. Data

While we assume that control information is passed along by signals while data is passed along as flow along the edges of a dataflow graph, the distinction between the two is really not as sharp as one might believe. Signals are a mechanism for modelling the generation of input/output events for state machines. However, there is really no intrinsic reason to distinguish between signals and data. It is basically an engineering decision to call certain types of flow signals and other types data. In terms of scheduling, the distinction is important because data flow that involves a choice of computation path (see the AND/OR model in a following section) imposes difficult scheduling problems. Sometimes we might be able to hide this type of choice by encapsulating them in a single functional module and using a signal to drive the execution of this module. However, the price to be payed here is that a module so chosen may have much more coarser grain. It is not clear in the global optimization how decisions to use signal or data affect the feasibility of building a system.

4.3 Object-Orientedness

Intuitively, system design can be simplified if individual state machines "exclusively determine" the timing constraints on the part of the dataflow graph they are associated with and likewise, the control signals are predicated only upon the values of local data flow. This might be thought of as the object-oriented equivalent of real-time system architecture. In practice, it is not clear how much this design guideline can actually buy us. There are still end-to-end timing assertions that may involve two or more "objects", and the encapsulation of data with control in the object-oriented approach says little about how end-to-end constraints ought to be taken into account in a decomposition. The bottom line is that while data and control flow may be more neatly handled by encapsulation into single modules, global timing constraint needs a more sophisticated concept of modularization. This is an important open problem.

In the next section, we discuss some details about the dataflow structure of distributed hard-real-time systems.

5 Dataflow Models

A basic dataflow model is introduced in [5]. An extended dataflow model (AND/OR model) was introduced in [6] for modelling a more general class of distributed hard real-time systems. Both dataflow models have been extensively studied for the case of uniprocessor systems. The models are for real-time systems where the execution of the functional elements is governed by the presence of data. The timing constraints for these models are introduced on the basis of the rate of the arriving data and the constraint that no-data-loss is permitted. When studying the distributed environment, several issues arise which are not present in the uniprocessor case, such as the task allocation problem, network communication scheduling and guaranteeing end-to-end timing constraints for the whole system. In the following, we describe the basic model and the extended model. The objective is to formulate the resource scheduling problems to motivate future research. It is not our purpose to discuss solution ideas here.

5.1 The Basic Model

We now introduce the formal definitions in our dataflow model.

Definition A *graph model* is a triple (G, f, h) where:

- $G = (V, E)$ is a directed acyclic graph.
- E is the set of directed edges communicating the nodes in the graph.
- $V = V_I \cup V_C$ is the set of nodes in the graph, which represent *functional elements* (software modules or hardware devices). $V_I \cap V_C = \emptyset$
- V_I is the set of *input elements*, which have no incoming edges. These nodes represent the source of periodic data arrival and they require no computing resources.
- V_C is the set of *computation elements*, which require a known bounded amount of processing time.
- h is a function that maps each edge (u, v) in G onto a nonnegative integer which is the size of the data item passing from node u to node v. The h function is useful for defining the inter-processor traffic load on a multiprocessor implementation of the model. For the uniprocessor case, the h function is irrelevant, since the cost of passing data can be included in the computation times of the computation elements.
- f is a function that maps every node in V to a list of integers and it is defined as follows for every node v_j:

$$f(v_j) = \begin{cases} c_j & \text{if } v_j \in V_C \\ (\theta_j, p_j) & \text{if } v_j \in V_I \end{cases}$$

- θ_j is the lag of v_i which is defined to be the time at which v_i becomes ready for execution for the first time. The lag in an input module is measured relative to the time at which the system starts running.
- c_j is the computation time for the functional module.
- p_j is the period of the input module and is defined to be the length of the time interval between two successive requests for executing the input module. (Each input module can be thought of as a hardware device which will periodically read from the external environment a data item or a message for the processor to process.)

Definition The *execution rules* for the basic dataflow model are described as follows.

Input rules:

1. Each input module must be executed once every period, i.e., the i^{th} execution of an input module v_j must occur at $\theta_j + (i-1)p_j$.
2. At the beginning of the execution of a computation module, it removes a data item (if any) from every one of its input edge(s).

Output rules:

1. When an input module is executed, it instantaneously outputs a data item on every one of its output edge(s), i.e., input modules take no computation time to execute.

2. At the completion of an execution of a computation module, it outputs a new data item on every one of its output edges, i.e., it behaves like an AND node. (Each edge e acts as a single buffer whose size is $h(e)$.) Any previous output data on the edges which have not been removed by a succeeding computation module will be replaced by more current data. When this happens, we say that *data loss* has occurred (also referred to as the *no-data-loss* requirement). Output data of a functional module are immediately available to the computation modules which are its immediate successors in the graph. The difference between an input element and a computation element in this regard is the time when their output become available.
3. A new data item is output from a computation node at the completion of each of its executions.

5.2 The Extended Model, the AND-OR Model

The need for an extension of the basic model defined in the previous section stems from the rather severe restriction on the output of the elements in a graph, i.e., each computation element is required to deposit its output data on all its outgoing edges. This restriction limits the practical application of the model for nontrivial systems in which structures such as selective branches and data-dependent computations are inherent and essential.

The extension of the model is performed by introducing an additional type of computation module, the output-OR modules (or simply, OR nodes), and by modifying the execution rules for the model. An OR node sends output to *zero* or more of its immediate successors after it completes its computation. Only computation elements can be either AND or OR type.

Definition The *execution rules* for the extended dataflow model are described as follows:

Input rules:

1. Each input module must be executed once every period, i.e., the i^{th} execution of an input module v_j must occur at $\theta_j + (i-1)p_j$.
2. At the beginning of the execution of a computation module, it removes a data item (if any) from every one of its input edge(s).

Output rules:

1. When an input module is executed, it instantaneously outputs a data item on every one of its output edge(s), i.e., input modules take no computation time to execute.
2. At the completion of an execution of an AND computation module, it outputs a new data item on every one of its output edges. A new data item is output from a computation node at the completion of each execution (i.e. the semantics are the same as in the basic model).
3. At the completion of an execution of an OR computation module, it outputs a new data item in *zero* or more of its output edges. It is possible therefore that an execution of an OR node may not fire any of its immediate predecessors.

6 Future Research

In this paper, we bring into focus some design issues concerning the integration of the control and dataflow structures of hard real-time systems. Quite clearly, there are many interesting open problems awaiting a solution when we consider parallel and distributed real-time systems. This paper hopefully serves as a starting point for further research in this area.

References

[1] R. Bettati, D. Gillies, C.C. Han, K.J. Lin, C.L. Liu, J.W.S. Liu, and W.K. Shih. Recent Results in Real-Time Scheduling. In A. van Tilborg and G. Koob, editors, *Foundations of Real-Time Computing: Scheduling and Resource Management*, chapter 4, pages 91–127. Kluwer Academic Publishers, 1991.

[2] D. Harel. “Statecharts: A Visual Formalism for Complex Systems”. Technical report, The Weizmann Institute of Science, July 1986.

[3] F. Jahanian and A.K. Mok. “Safety Analysis of Timing Properties of Real-Time Systems”. *IEEE Transactions on Software Engineering*, 12(9):890–904, September 1986.

[4] A. Mok. Towards Mechanization of Real-Time System Design. In A. van Tilborg and G. Koob, editors, *Foundations of Real-Time Computing: Formal Specifications and Methods*, chapter 1, pages 1–38. Kluwer Academic Publishers, 1992.

[5] A. Mok and S. Sutanthavibul. “Modeling and Scheduling of Dataflow Real–Time Systems”. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 178–187, December 1985.

[6] S. Sutanthavibul. *“A Data-Flow Model of Real-Time Systems”*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, May 1990.