End-to-End Design of Real-Time Systems *

Richard Gerber, Dong-in Kang
Dept. of Computer Science
University of Maryland
College Park, MD 20742
{rich,dikang}@cs.umd.edu

Seongsoo Hong
Silicon Graphics Inc.
2011 N. Shoreline Blvd.
Mountain View, CA 94039
sshong@engr.sgi.com

Manas Saksena
Dept. of Computer Science
Concordia University
Montreal, Quebec H3G 1M8, Canada
manas@cs.concordia.ca

April 20, 1995

Abstract

This chapter presents a comprehensive design methodology for guaranteeing end-to-end requirements of real-time systems. Applications are structured as a set of process components connected by asynchronous channels, in which the endpoints are the system's external inputs and outputs. Timing constraints are then postulated between these inputs and outputs; they express properties such as end-to-end propagation delay, temporal input-sampling correlation, and allowable separation times between updated output values.

The automated design method works as follows: First new tasks are created to correlate related inputs, and an optimization algorithm, whose objective is to minimize CPU utilization, transforms the end-to-end requirements into a set of intermediate rate constraints on the tasks. If the algorithm fails, a restructuring tool attempts to eliminate bottlenecks by transforming the application, which is then re-submitted into the assignment algorithm. The final result is a schedulable set of fully periodic tasks, which collaboratively maintain the end-to-end constraints.

*This research is supported in part by ONR grant N00014-94-10228, NSF grant CCR-9209333, and NSF Young Investigator Award CCR-9357850.

1 Introduction

Most real-time systems possess only a small handful of *inherent* timing constraints which will "make or break" their correctness. These are called *end-to-end* constraints, and they are established on the systems' external inputs and outputs. Two examples are:

(1) *Temperature updates rely on pressure and temperature readings correlated within 10μs.*

(2) *Navigation coordinates are updated at a minimum rate of 40ms, and a maximum rate 80ms.*

But while such end-to-end timing parameters may indeed be few in number, maintaining *functionally correct* end-to-end values may involve a large set of interacting components. Thus, to ensure that the end-to-end constraints are satisfied, each of these components will, in turn, be subject to their own *intermediate* timing constraints. In this manner a small handful of end-to-end constraints may – in even a modest system – yield a great many intermediate constraints.

The task of imposing timing parameters on the functional components is a complex one, and it mandates some careful engineering. Consider example (2) above. In an avionics system, a "navigation update" may require such inputs as "current heading," airspeed, pitch, roll, etc; each sampled within varying degrees of accuracy. Moreover, these attributes are used by other subsystems, each of which imposes its own tolerance to delay, and possesses its own output rate. Further, the navigation unit may itself have other outputs, which may have to be delivered at rates faster than 40ms, or perhaps slower than 80ms. And to top it off, subsystems may share limited computer resources. A good engineer balances such factors, performs extensive trade-off analysis, simulations and sensitivity analysis, and proceeds to assign the constraints.

These intermediate constraints are inevitably on the conservative side, and moreover, they are conveyed to the programmers in terms of constant values. Thus a scenario like the following is often played out: The design engineers mandate that functional units A, B and C execute with periods 65ms, 22ms and 27ms, respectively. The programmers code up the system, and find that C grossly over-utilizes its CPU; further, they discover that most of C's outputs are not being read by the other subsystems. And so, they go back to the engineers and "negotiate" for new periods – for example 60ms, 10ms and 32ms. This process may continue for many iterations, until the system finally gets fabricated.

This scenario is due to a simple fact: the end-to-end requirements allow many possibilities for the intermediate constraints, and engineers make what they consider to be a rational selection. However, the basis for this selection can only include rough notions of software structuring and scheduling policies – after all, many times the hardware is not even fabricated at this point!

Our Approach. In this chapter we present an alternative strategy, which maintains the timing constraints in their end-to-end form for as long as possible. Our design method iteratively instantiates the intermediate constraints, all the while taking advantage of the leeway inherent in the end-to-end constraints. If the assignment algorithm fails to produce a full set of intermediate constraints, potential bottlenecks are identified. At this point an application analysis tool takes over, determines potential solutions to the bottleneck, and if possible, restructures the application to avoid it. The result is then re-submitted into the assignment algorithm.

We have implemented a significant portion of our approach as part of integrated design tool development effort at the University of Maryland. The tool, named TimeWare/DesignAssistant, graphically and textually captures both a system design and its end-to-end requirements, and then produces intermediate constraints. Throughout the chapter, we use examples we take from the tool's graphical interface.

Scope of Examples. Due to the complexity of the general problem, in this chapter we confine our discussion to systems possessing the following characteristics.

1: We assume our applications possess three classes of timing constraints which we call *freshness*, *correlation* and *separation*.

- A *freshness constraint* (sometimes called propagation delay) bounds the time it takes for data to flow through the system. For example, assume that an external output Y is a function of some system input X. Then a freshness relationship between X and Y might be: "If Y is delivered at time t, then the X-value used to compute Y is sampled no earlier than $t - 10$ms." We use the following notation to denote this constraint: "$F(Y|X) = 10$."
- A *correlation constraint* limits the maximum time-skew between several inputs used to produce an output. For example, if X_1 and X_2 are used to produce Y, then a correlation relationship may be "if Y is delivered at time t, then the X_1 and X_2 values used to compute Y are sampled no more than within 2ms of each other." We denote this constraint as "$C(Y|X_1, X_2) = 2$."
- A *separation constraint* constrains the jitter between consecutive values on a single output channel, say Y. For example, "Y is delivered at a minimum rate of 3ms, and a maximum rate of 13ms," denoted as $l(Y) = 3$ and $u(Y) = 13$, respectively.

While this constraint classification is not complete, it is sufficiently powerful to represent many timing properties one finds in a requirements document. (Our initial examples (1) and (2) are correlation and separation constraints, respectively.) Note that a single output Y_1 may – either directly or indirectly – be subject to several interdependent constraints. For example, Y_1 might require tightly correlated inputs, but may abide with relatively lax freshness constraints. However, perhaps Y_1 also requires data from an intermediate subsystem which is, in turn, shared with a very high-rate output Y_2.

2: Subsystems execute on a single CPU. Our approach can be extended for use in distributed systems, a topic we revisit in Section 8. For the sake of presenting the intermediate constraint-assignment technique, in this chapter we limit ourselves to uniprocessor systems.

3: The entity-relationships within a subsystem are already specified. For example, if a high-rate video stream passes through a monolithic, compute-intensive filter task, this situation may easily cause a bottleneck. If our algorithm fails to find a proper intermediate timing constraint for the filter, the tool will attempt to restructure it to optimize it as much as possible. In the end, however, it cannot redesign the system.

Finally, we stress that we are not offering a *completely automatic* solution. Even with a fully periodic task model, assigning periods to the intermediate components is a complex, nonlinear optimization problem which – at worst – can become combinatorially expensive. As for software restructuring, the specific tactics used to remove bottlenecks will often require user interaction.

Problem and Solution Strategy. We note the above restrictions, and tackle the intermediate constraint-assignment problem, as rendered by the following ingredients:

- A set of external inputs $\{X_1, \ldots, X_n\}$, outputs $\{Y_1, \ldots, Y_m\}$, and the end-to-end constraints between them.
- A set of intermediate component tasks $\{P_1, \ldots, P_l\}$.
- A task graph, denoting the communication paths from the inputs, through the tasks, and to outputs.

Solving the problem requires setting timing constraints for the intermediate components, so that all end-to-end constraints are met. Moreover, during any interval of time utilization may never exceed 100%.

Our solution employs the following ingredients: (1) A periodic, preemptive tasking model (where it is the our algorithm's duty to assign the rates); (2) a buffered, asynchronous communication scheme, allowing us to keep down IPC times; (3) the period-assignment, optimization algorithm, which forms the heart of the approach; and (4) the software-restructuring tool, which takes over when period-assignment fails.

Related Work. This research was, in large part, inspired by the real-time transaction model proposed by Burns *et. al.* in [3]. While the model was formulated to express database applications, it can easily incorporate variants of our *freshness* and *correlation* constraints. In the analogue to freshness, a persistent object has "absolute consistency within t" when it corresponds to real-world samples taken within maximum drift of t. In the analogue to correlation, a set of data objects possesses "relative consistency within t" when all of the set's elements are sampled within an interval of time t.

We believe that in output-driven applications of the variety we address, separation constraints are also necessary. Without postulating a minimum rate requirement, the freshness and correlation constraints can be vacuously satisfied – by never outputting any values! Thus the separation constraints enforce the system's progress over time.

Burns *et. al.* also propose a method for deriving the intermediate constraints; as in the data model, this approach was our departure point. Here the high-level requirements are re-written as a set of constraints on task periods and deadlines, and the transformed constraints can hopefully be solved. There is a big drawback, however: the correlation and freshness constraints can inordinately tighten deadlines. E.g., if a task's inputs must be correlated within a very tight degree of accuracy – say, several nanoseconds – the task's deadline has to be tightened accordingly. Similar problems accrue for freshness constraints. The net result may be an over-constrained system, and a potentially unschedulable one.

Our approach is different. With respect to tightly correlated samples, we put the emphasis on simply getting the data into the system, and then passing through in due time. However, since this in turn causes many different samples flowing through the system at varying rates, we perform "traffic control" via a novel use of "virtual sequence numbering." This results in significantly looser periods, constrained mainly by the freshness and separation requirements. We also present a period assignment problem which is optimal – though quite expensive in the worst case.

This work was also influenced by Jeffay's "real-time producer/consumer model" [11], which possesses a task-graph structure similar to ours. In this model rates are chosen so that all messages "produced" are eventually "consumed." This semantics leads to a tight coupling between the execution of a consumer to that of its producers; thus it seems difficult to accommodate relative constraints such as those based on freshness.

Klein *et. al.* surveys the current engineering practice used in developing industrial real-time systems [12]. As is stressed, the intermediate constraints should be primarily a function of the end-to-end constraints, but should, if possible, take into account sound real-time scheduling techniques. At this point, however, the "state-of-the-art" is the practice of trial and error, as guided by engineering experience. And this is exactly the problem we address in this chapter.

Organization of this Chapter. The remainder of the chapter is organized as follows. In Section 2 we introduce the application model and formally define our problem. In Section 3 we show our method of transforming the end-to-end constraints into intermediate constraints on the tasks. In Section 4 we describe the constraint-solver in detail, and push through a small example. In Section 5 we describe the application transformer, and in Section 6 we show how the executable application is finally built. In Section 7 we discuss the prototype implementation of our tool.

2 Problem Description and Overview of Solution

We re-state our problem as follows:

- Given a task graph with end-to-end timing constraints on its inputs and outputs,
- Derive periods, offsets and deadlines for every task,
- Such that the end-to-end requirements are met.

In this section we define these terms, and present the techniques behind our solution strategy. We also privide an overview of our tool, named the TimeWare/DesignAssistant, which is based on the on the solutions described in this chapter. The tool consists of several components (see Figure 1), including an interactive, graphical interface for structuring the system componenets, and a set of toolbox functions which help automate the assignment of the intermediate process constraints.

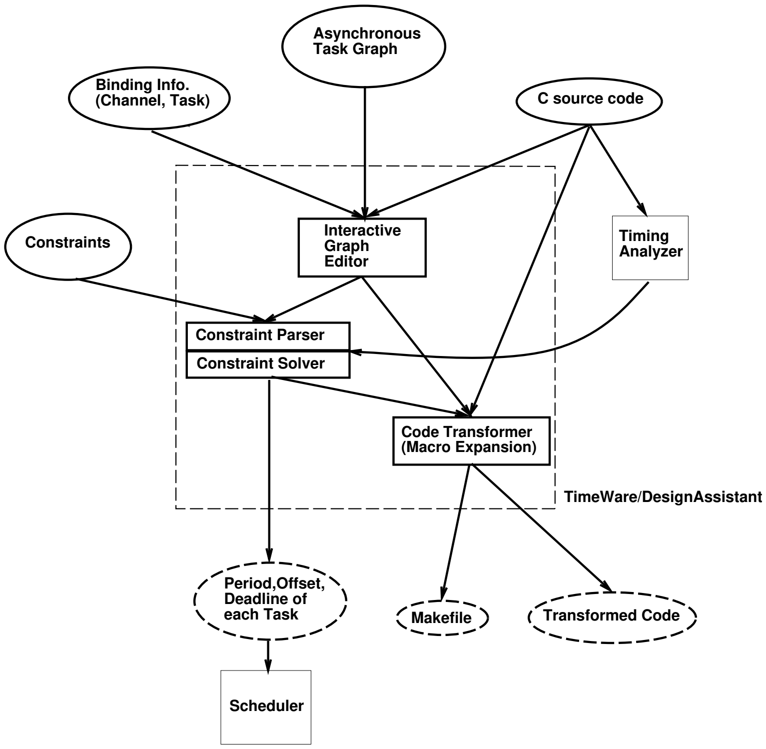


Figure 1: The structure of the TimeWare/DesignAssistant

2.1 The Asynchronous Task Graph

An application is rendered in an asynchronous task graph (ATG) format. Figure 2(A) shows an example ATG, drawn using the TimeWare/DesignAssistant interface. In general, an ATG $G(V, E)$ possesses the following attributes.

- $V = \mathcal{P} \cup \mathcal{D}$, where $\mathcal{P} = \{P_1, \ldots, P_n\}$, i.e., the set of tasks; and $\mathcal{D} = \{d_1, \ldots, d_m\}$, a set of asynchronous, buffered channels. In Figure 2(A) tasks are drawn as a circles around their associated names. The buffered channels are drawn as small rectangles, the inputs as white bold rectangles and the outputs as gray bold rectangles. We note that the external outputs and inputs are simply typed nodes in D.
- $E \subseteq (\mathcal{P} \times \mathcal{D}) \cup (\mathcal{D} \times \mathcal{P})$ is a set of directed edges, such that if $P_i \rightarrow d_j$ and $P_l \rightarrow d_j$ are both in E, then $P_i = P_l$. That is, each channel has a single-writer/multi-reader restriction.
- All $P_i \in \mathcal{P}$ have the following attributes: a period T_i, an offset $O_i \geq 0$ (denoting the earliest start-time from the start-of-period), a deadline $D_i \leq T_i$ (denoting the latest finish-time relative to the start-of-period), and a maximum execution time e_i. The interval $[O_i, D_i]$ constrains the window W_i of execution, where $W_i = D_i - O_i$.

Note that initially the T_i's, O_i's and D_i's are open variables, and they get instantiated by the constraint-solver.

The semantics of an ATG is as follows. Whenever a task P_i executes, it reads data from all incoming channels d_j corresponding to the edges $d_j \rightarrow P_i$, and writes to all channels d_l corresponding to the edges $P_i \rightarrow d_l$. The actual ordering imposed on the reads and writes is inferred by the task P_i's structure.

The tool binds these abstract task and channel names to real code. Consider the ATG in Figure 2(A), whose node P_4 is "blown up" in Figure 2(B). As the Figure 2(B,Top) shows, the function "foo" in the file "code.c" is bound to the node "P4." The programmer must also bind the abstract channel names to the corresponding identifiers in the module. The lower window of the Figure 2(B) shows the C code within code.c, and the stylistic conventons used for channel binding.

As far as the programmer is concerned the task P_4 has a (yet-to-be-determined) period T_4, and a set of asynchronous channels, accessible via generic operations such as "**Read**" and "**Write**." All reads and writes on channels are asynchronous and non-blocking. While a writer always inserts a value onto the end of the channel, a reader can (and many times will) read data from any location. For example, perhaps a writer runs at a period of 20ms, with two readers running at 120ms and 40ms, respectively. The first reader may use every sixth value (and neglect the others), whereas the second reader may use every other value.

But this scheme raises a "chicken and egg" issue, one of many that we faced in this work. One of our objectives is to support software reuse, in which functional components may be deployed in different systems – and have their timing parameters automatically calibrated to the physical limitations of each. But this objective would be hindered if a designer had to employ the following tedious method: (1) to *first* run the constraint-solver, which would find the T_i's, and *then*, based on the results; (2) to hand-patch all of the modules with specialized IPC code, ensuring that the intermediate tasks correctly correlate their input samples.

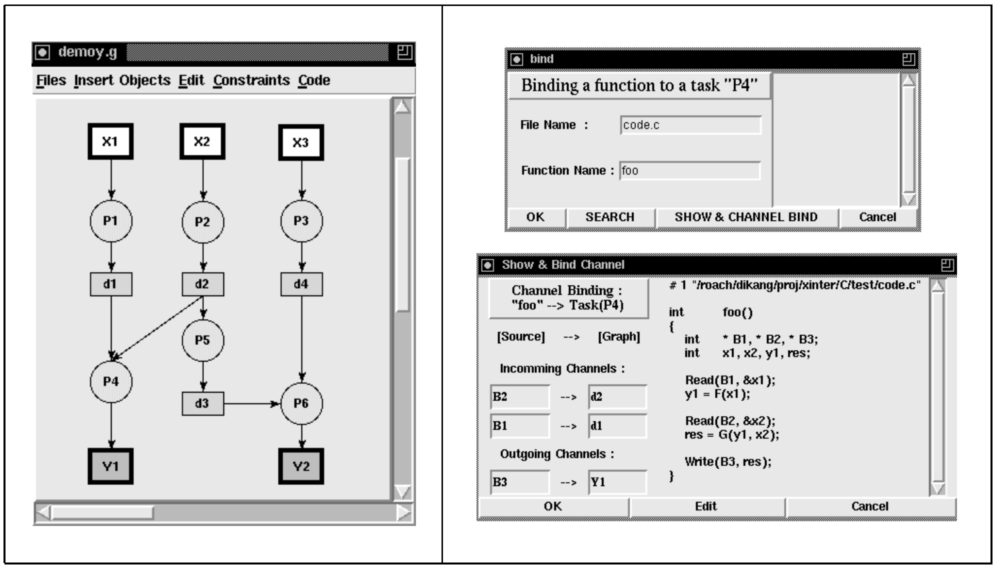
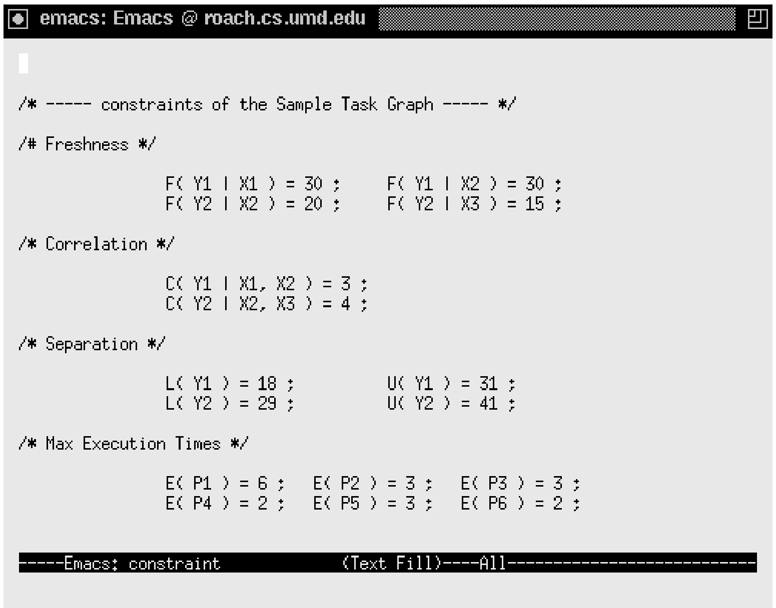


Figure 2: (A) A task graph and (B) code for P_4.



emacs: Emacs @ roach.cs.umd.edu

```
/* ----- constraints of the Sample Task Graph ----- */

/* Freshness */

            F( Y1 | X1 ) = 30 ;     F( Y1 | X2 ) = 30 ;
            F( Y2 | X2 ) = 20 ;     F( Y2 | X3 ) = 15 ;

/* Correlation */

            C( Y1 | X1, X2 ) = 3 ;
            C( Y2 | X2, X3 ) = 4 ;

/* Separation */

            L( Y1 ) = 18 ;          U( Y1 ) = 31 ;
            L( Y2 ) = 29 ;          U( Y2 ) = 41 ;

/* Max Execution Times */

            E( P1 ) = 6 ;   E( P2 ) = 3 ;   E( P3 ) = 3 ;
            E( P4 ) = 2 ;   E( P5 ) = 3 ;   E( P6 ) = 2 ;

-----Emacs: constraint          (Text Fill)----All--------------------------
```

Figure 3: Constraints description file of the sample task graph

We solve this problem as follows: after the constraint-assignment algorithm determines the task rates, a post-processing phase determines the actual space required for each channel. Then they are automatically implemented as circular, slotted buffers. With the channel size information, the tool's Code Transformer automatically generates code to allocate and initialize each channel. Then it patches the user's C code, instantiating each "**Read**" and "**Write**" operation to select the correct input value.

This type of scheme allows us to minimize the overhead incurred when blocking communication is used, and to concentrate exclusively on the assignment problem. In fact – as we show in the sequel – communication can be *completely unconditional*, in that we do not even require short locking for consistency. However, we pay a price for avoiding this overhead; namely, that the period assignments must ensure that no writer can overtake a reader currently accessing its slot.

Moreover, we note that our timing constraints define a system driven by time and *output requirements*. This is in contrast to reactive paradigms such as ESTEREL [4], which are input-driven. Analogous to the "conceptually infinite buffering" assumptions, the rate assignment algorithm assumes that the external inputs are always fresh and available. The *derived* input-sampling rates then determine the *true* requirements on input-availability. And since an input X can be connected to another ATG's output Y, these requirements would be imposed on Y's timing constraints.

2.2 A Small Example

As a simple illustration, consider again the system whose ATG is shown in Figure 2(A). It is composed of six interacting tasks with three external inputs and two external outputs. Figure 3 shows the application's end-to-end constraints, which the DesignAssistant treats as attributes of the ATG at hand.

While the system is small, it serves to illustrate several facets of the problem: (1) There may be many possible choices of rates for each task; (2) correlation constraints may be tight compared to the allowable end-to-end delay; (3) data streams may be shared by several outputs (in this case that originating at X_2); and (4) outputs with the tightest separation constraints may incur the highest execution-time costs (in this case Y_1, which exclusively requires P_1).

2.3 Problem Components

Guaranteeing the end-to-end constraints actually poses three sub-problems, which we define as follows.

Correctness: Let $\mathcal{C}$ be the set of derived, intermediate constraints and $\mathcal{E}$ be the set of end-to-end constraints. Then all system behaviors that satisfy $\mathcal{C}$ also satisfy $\mathcal{E}$.

Feasibility: The task executions inferred by $\mathcal{C}$ never demand an interval of time during which utilization exceeds 100%.

Schedulability: There is a scheduling algorithm which can efficiently maintain the intermediate constraints $\mathcal{C}$, and preserve feasibility.

In the problem we address, the three issues cannot be decoupled. Correctness, for example, is often treated as verification problem using a logic such as RTL [10]. Certainly, given the ATG we could formulate $\mathcal{E}$ in RTL and query whether the constraint set is satisfiable. However, a "yes" answer would give us little insight into finding a good choice for $\mathcal{C}$ – which must, after all, be simple enough to schedule. Or, in the case of methods like model-checking ([1], etc.), we could determine whether $\mathcal{C} \Rightarrow \mathcal{E}$ is invariant with respect to the system. But again, this would be an *a posteriori* solution, and assume that we already possess $\mathcal{C}$. On the other hand, a system that is feasible may still not be schedulable under a *known* algorithm; i.e., one that can be efficiently managed by a realistic kernel.

In this chapter we put our emphasis on the first two issues. However, we have also imposed a task model for which the greatest number of efficient scheduling algorithms are known: simple, periodic dispatching with offsets and deadlines. In essence, by restricting $\mathcal{C}$'s free variables to the T_i's, O_i's and D_i's, we ensure that feasible solutions to $\mathcal{C}$ can be easily checked for schedulability.

The problem of scheduling a set of periodic real-time tasks on a single CPU has been studied for many years. Such a task set can be dispatched by a calendar-based, non-preemptive schedule (e.g., [18, 19, 20]), or by a preemptive, static-priority scheme (e.g., [5, 13, 15, 17]). For the most part our results are independent of any particular scheduling strategy, and can be used in concert with either non-preemptive or preemptive dispatching.

However, in the sequel we frequently assume an underlying static-priority architecture. This is for two reasons. First, a straightforward priority assignment can often capture most of the ATG's precedence relationships, which obviates the need for superfluous offset and deadline variables. Thus the space of feasible solutions can be simplified, which in turn reduces the constraint-solver's work. Second, priority-based scheduling has recently been shown to support all of the ATG's inherent timing requirements: pre-period deadlines [2], precedence constrained sub-tasks [9], and offsets [16]. A good overview to static priority scheduling may be found in [5].

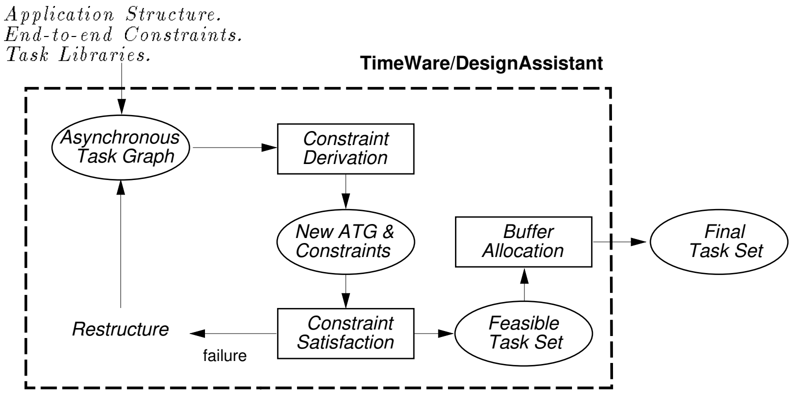


Figure 4: Overview of the approach.

2.4 Overview of the Solution

Our solution is carried out in a four-step process, as shown in Figure 4. In **Step 1**, the intermediate constraints $\mathcal{C}$ are derived, which postulates the periods, deadlines and offsets as free variables. The challenge here is to balance several factors – correctness, feasibility and simplicity. That is, we require that any solution to $\mathcal{C}$ will enforce the end-to-end constraints $\mathcal{E}$, and that any solution must also be feasible. At the same time, we want to keep $\mathcal{C}$ as simple as possible, and to ensure that finding a solution is a relatively straightforward venture. This is particularly important since the feasibility criterion – defined by CPU utilization – introduces non-linearities into the constraint set. In balancing our goals we impose additional structure on the application; e.g., by creating new sampler tasks to get tightly correlated inputs into the system.

In **Step 2** the constraint-solver finds a solution to $\mathcal{C}$, which is done in several steps. First $\mathcal{C}$ is solved for the period variables, the T_i's, and then the resulting system is solved for the offsets and deadlines. Throughout this process we use several heuristics, which exploit the ATG's structure.

If a solution to $\mathcal{C}$ cannot be found, the problem often lies in the original design itself. For example, perhaps a single, stateless server handles inputs from multiple clients, all of which run at wildly different rates. **Step 3**'s restructuring tool helps the programmer eliminate such bottlenecks, by automatically replicating strategic parts of the ATG.

In **Step 4**, the derived rates are used to reserve memory for the channels, and to instantiate the "**Read**" and "**Write**" operations. For example, consider P_4 in Figure 2(A), which reads from channels d_1 and d_2.

Now, assume that the constraint-solver assigns P_4 and P_2 periods of 30ms and 10ms, respectively. Then P_4's **Read** operation on d_2 would be replaced by a macro, which would read every third data item in the buffer – and would skip over the other two.

Harmonicity. The above scheme works only if a producer can always ensure that it is not overtaking its consumers, and if the consumers can always determine which data item is the correct one to read. For example, P_4's job in managing d_2 is easy – since T_2 = 10ms and T_4 = 30ms, P_4 will read every third item out of the channel.

But P_4 has another input channel, d_1; moreover, temporally correlated samples from the two channels have to be used to produce a result. What would happen if the solver assigned P_1 a period of 30ms, but gave P_2 a period of 7ms?

If the tasks are scheduled in rate-monotonic order, then d_2 is filled five times during P_4's first frame, four times during the second frame, etc. In fact since 30 and 7 are relatively prime, P_4's selection logic to correlate inputs would be rather complicated. One solution would be to time-stamp each input X_1 and X_2, and then pass these stamps along with all intermediate results. But this would assume access to a precise hardware timer; moreover, time-stamps for multiple inputs would have to be composed in some manner. Worst of all, each small data value (e.g., an integer) would carry a large amount of reference information.

The obvious solution is the one that we adopt: to ensure that every "chain" possesses a common base clock-rate, which is exactly the rate of the task at the head of the chain. In other words, we impose a harmonicity constraint between (producer, consumer) pairs; (i.e., pairs (P_p, P_c) where there are edges $P_p \rightarrow d$ and $d \rightarrow P_c$.)

Definition 2.1 (Harmonicity) *A task P_2 is harmonic with respect to a task P_1 if T_2 is exactly divisible by T_1 (represented as $T_2|T_1$[1]).*

Consider Figure 2(A), in which there are three chains imposing harmonic relationships. In this tightly coupled system we have that $T_4|T_1$, $T_4|T_2$, $T_5|T_2$, $T_6|T_5$ and $T_6|T_3$.

3 Step 1: Deriving the Constraints

In this section we show the derivation process of intermediate constraints, and how they (conservatively) guarantee the end-to-end requirements. We start the process by synthesizing the intermediate correlation constraints, and then proceed to treat freshness and separation.

3.1 Synthesizing Correlation Constraints

Let's revisit our example task graph (now in Figure 5(A)), where the three inputs X_1, X_2 and X_3 are sampled by three separate tasks. If we wish to guarantee that P_1's sampling of X_1 is correctly correlated to P_2's sampling of X_2, we must pick short periods for both P_1 and P_2. Indeed, in many practical real-time systems, the correlation requirements may very well be tight, and way out of proportion with the freshness constraints. This typically results in periods that get tightened exclusively to accommodate correlation, which can easily lead to gross over-utilization. Engineers often call this problem "over-sampling," which is somewhat of a misnomer, since sampling rates may be tuned expressly for coordinating inputs. Instead, the problem arises from poor coupling of the sampling and computational activities.

[1] $x|y$ iff $\exists \alpha ::\ \alpha y = x$ and $\alpha \geq 1$, where α is an integer.

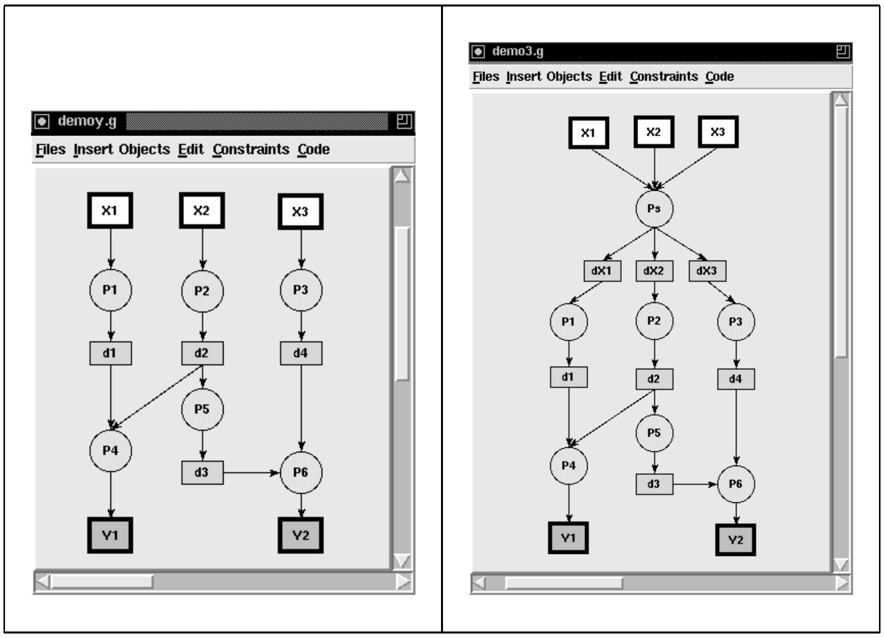


Figure 5: (A) Original task graph and (B) transformed task graph.

Thus our approach is to decouple these components as much as possible, and to create specialized samplers for related inputs. For a given ATG, the sampler derivation is performed in the following manner.

foreach Correlation constraint $C_l(Y_k|X_{l_1}, \ldots, X_{l_m})$
  Create the set of all input-output pairs associated with C_l, i.e.,
   $T_l := \{(X_{l_i}, Y_k)|\, X_{l_i} \in \{X_{l_1}, \ldots, X_{l_m}\}\}$
foreach T_l, **foreach** T_k
  If there's a common input X such that there exist outputs Y_i, Y_j
   with $(X, Y_i) \in T_l$, $(X, Y_j) \in T_k$, and
  if chains from X to Y_l and X to Y_k share a common task, then
    Set $T_l := T_l \cup T_k$; $T_k := \emptyset$
foreach T_l, identify all associated sampling tasks, i.e.,
  $S_l := \{P|(X, Y) \in T_l \wedge X \rightarrow P\}$
  If $|S_l| > 1$, create a periodic sampler P_{s_l} to take samples for inputs in T_l

Thus the incoming channels from inputs T_l to tasks in S_l are "intercepted" by the new sampler task P_{s_l}.

Returning to our original example, which we repeat in Figure 5(A). Since both correlated inputs share the center stream, the result is a single group of correlated inputs $\{(X_1, X_2, X_3)\}$. This, in turn, results in the formation of the single sampler P_s. We assume P_s has a low execution cost of 1. The new, transformed graph is shown at the right column of Figure 5(B).

As for the deadline-offset requirements, a sampler P_{s_l} is constrained by the following trivial relationship

$$D_{s_l} - O_{s_l} \leq t_{cor}$$

where t_{cor} is the maximum allowable time-drift on all correlated inputs read by P_{s_l}.

The sampler tasks ensure that correlated inputs are read into the system within their appropriate time bounds. This allows us to solve for process rates as a function of both the freshness and separation constraints, which vastly reduces the search space.

However we cannot ignore correlation altogether, since merely sampling the inputs at the same time does not guarantee that they will *remain* correlated as they pass through the system. The input samples may be processed by different streams (running at different rates), and thus they may still reach their join points at different absolute times.

For example, refer back to Figure 5, in which $F(Y_2|X_2) > F(Y_2|X_3)$. This disparity is the result of an under-specified system, and may have to be tightened. The reason is simple: if P_6's period is derived by using correlation as a dominant metric, the resulting solution may violate the tighter freshness constraints. On the other hand, if freshness is the dominant metric, then the correlation constraints may not be achieved.

We solve this problem by eliminating the "noise" that exists between the different set of requirements. Thus, whenever a fresh output is required, we ensure that there are correlated data sets to produce it. In our example this leads to tightening the original freshness requirement $F(Y_2|X_2)$ to $F(Y_2|X_3)$.

Thus we invoke this technique as a general principle. For an output Y with correlated input sets $X_1, \ldots, X_m$, the associated freshness constraints are adjusted accordingly:

$$F(Y|X_1), \ldots, F(Y|X_m) \ := min\{F(Y|X_1), \ldots, F(Y|X_m)\}$$

3.2 Synthesizing Freshness Constraints

Consider a freshness constraint $F(Y|X) = t_f$, and recall its definition:

> *For every output of Y at some time t, the value of X used to compute Y must have been read no earlier that time $t - t_f$.*

As data flows through a task chain from X to Y, each task P adds two types of delay overhead to the data's end-to-end response time. One type is *execution time*, i.e., the time required for P to process the data, produce outputs, etc. In this chapter we assume that P's maximum execution time is fixed, and has already been optimized as much as possible by a good compiler.

The other type of delay is *transmission latency*, which is imposed while P waits for its correlated inputs to arrive for processing. Transmission time is not fixed; rather, it is largely dependent on our derived process-based constraints. Thus minimizing transmission time is our goal in achieving tight freshness constraints.

Fortunately, the harmonicity relationship between producers and consumers allows us to accomplish this goal. Consider a chain $P_1, P_2, \ldots, P_n$, where P_n is the output task, and P_1 is the input

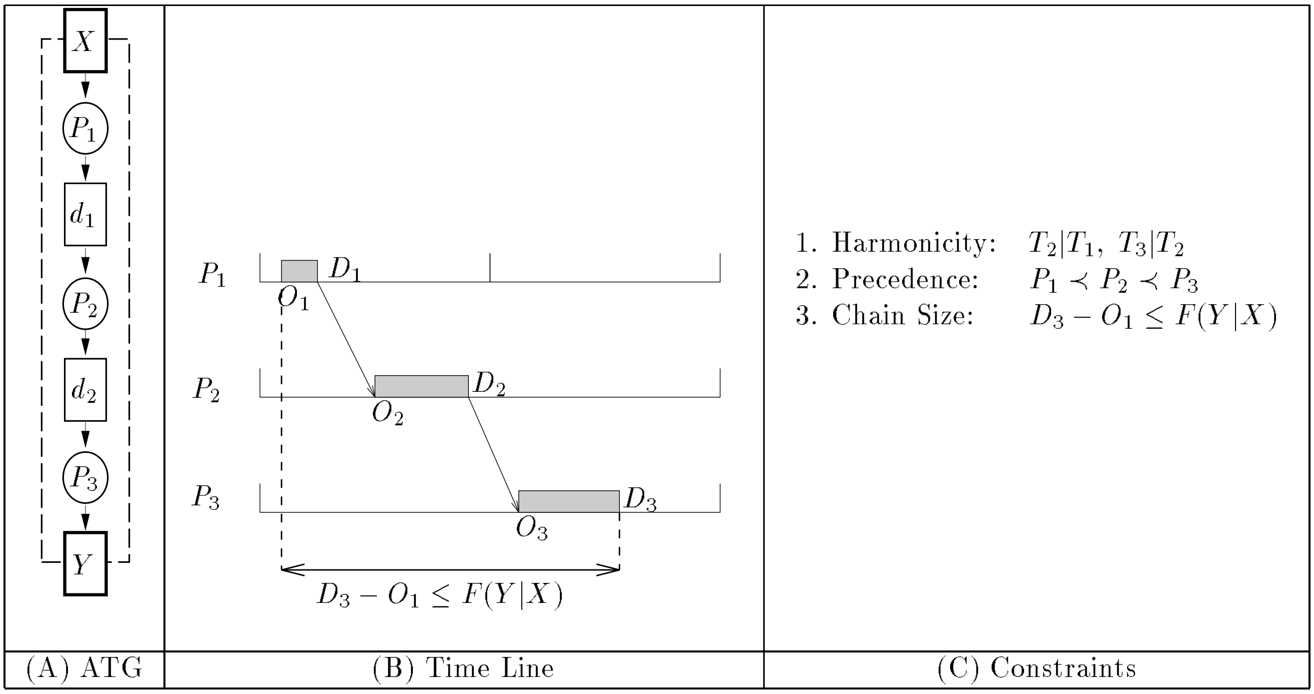


Figure 6: Freshness constraints with coupled tasks.

task. From the harmonicity constraints we get $T_{i+1}|T_i$, for $1 \leq i < n$. Assuming that all tasks are started at time 0, whenever there is an invocation of the output task P_n, there are simultaneous invocations of every task in the freshness chain.

Consider Figure 6 in which there are three tasks P_1, P_2 and P_3 in a freshness chain. From the harmonicity assumption we have $T_3|T_2$ and $T_2|T_1$.

The other constraints are derived for the entire chain, under the scenario that within each task's minor frame, input data gets read in, it gets processed, and output data is produced. Under these constraints, the worst case end-to-end delay is given by $D_n - O_1$, and the freshness requirement is guaranteed if the following holds:

$$D_n - O_1 \leq t_f$$

Note that we also require a precedence between each producer/consumer task pair. As we show in Figure 6, this can be accomplished via the offset and deadline variables – i.e., by mandating that $D_i \leq O_{i+1}$, for $1 \leq i < n$.

But this approach has the following obvious drawback: *The end-to-end freshness t_f must be divided into fixed portions of slack at each node.* On a global system-wide level, this type of rigid flow control is not the best solution. It is not clear how to distribute the slack between intermediate tasks, without over-constraining the system. More importantly, with a rigid slack distribution, a

$F(Y_1 \vert X_1)$	$F(Y_1 \vert X_2)$	$F(Y_2 \vert X_2)$	$F(Y_2 \vert X_3)$
$D_4 - O_s \leq 30$	$D_4 - O_s \leq 30$	$D_6 - O_s \leq 15$	$D_6 - O_s \leq 15$
$O_s + e_s + e_1 \leq D_1$	$O_s + e_s + e_2 \leq D_2$	$O_s + e_s + e_2 + e_5 \leq D_5$	$O_s + e_s + e_3 \leq D_3$
$D_1 \leq O_4$	$D_2 \leq O_4$	$D_5 \leq O_6$	$D_3 \leq O_6$
$T_4 \vert T_1, \quad T_1 \vert T_s$	$T_4 \vert T_2, \quad T_2 \vert T_1$	$T_6 \vert T_5, \quad T_5 \vert T_2, \quad T_2 \vert T_s$	$T_6 \vert T_3, \quad T_3 \vert T_s$

Table 1: Constraints due to freshness requirements.

consumer task would not be allowed to execute before its offset, *even if its input data is available.*[2]

Rather, we make a straightforward priority assignment for the tasks in each chain, and let the scheduler enforce the precedence between them. In this manner, we can do away with the intermediate deadline and offset variables. This leads to the following rule of thumb:

> *If the consumer task is not the head or tail of a chain, then its precedence requirement is deferred to the scheduler. Otherwise, the precedence requirement is satisfied through assignment of offsets.*

Example. Consider the freshness constraints for our example in Figure 5(A), $F(Y_1|X_1) = 30$, $F(Y_1|X_2) = 30$, $F(Y_2|X_2) = 15$, and $F(Y_2|X_3) = 15$. The requirement $F(Y_1|X_1) = 30$ specifies a chain window size of $D_4 - O_s \leq 30$. Since P_1 is an intermediate task we now have the precedence $P_s \prec P_1$, which will be handled by the scheduler. However, according to our "rule of thumb," we use the offset for P_4 to handle the precedence $P_1 \prec P_4$. This leads to the constraints $D_1 \leq O_4$ and $D_s \leq D_1 - e_1$. Similar inequalities are derived for the remaining freshness constraints, the result of which is shown in Table 1.

3.3 Output Separation Constraints

Consider the separation constraints for an output Y, generated by some task P_i. As shown in Figure 7, the window of execution defined by O_i and D_i constrains the time variability within a period. Consider two frames of P_i's execution. The widest separation for two successive Y's can occur when the first frame starts as early as possible, and the second starts as late as possible. Conversely, the opposite situation leads to the smallest separation.

Thus, the separation constraints will be satisfied if the following holds true:

$$(T_i + D_i) - O_i \leq u(Y) \quad \text{and} \quad (T_i - D_i) + O_i \geq l(Y)$$

[2]Note that corresponding issues arise in real-time rate-control in high-speed networks.

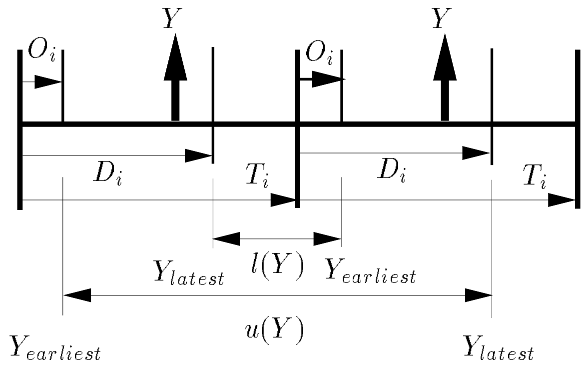


Figure 7: Separation constraints for two frames.

Example. Consider the constraints that arise from output separation requirements, which are induced on the output tasks P_4 and P_6. The derived constraints are presented below:

$$\begin{array}{ll} (T_4 + D_4) - O_4 \leq u(Y_1) & (T_4 - D_4) + O_4 \geq l(Y_1) \\ (T_6 + D_6) - O_6 \leq u(Y_2) & (T_6 - D_6) + O_6 \geq l(Y_2) \end{array}$$

3.4 Execution Constraints:

Clearly, each task needs sufficient time to execute. This simple fact imposes additional constraints, that ensure that each task's maximum execution time can fit into its window. Recall that (1) we use offset, deadline and period variables for tasks handling external input and output; and (2) we use period variables and precedence constraints for the intermediate constraints.

We can easily preserve these restrictions when dealing with execution time. For each external task P_i, the following inequalities ensure that window-size is sufficiently large for the CPU demand:

$$O_i + e_i \ \leq \ D_i, \quad O_i \geq 0 \quad D_i \leq T_i$$

On the other hand, the intermediate tasks can be handled by imposing restrictions on their constituent chains. For a single chain, let E denote the chain's total execution time. Then the chain-wise execution constraints are:

$$O_h + E \ \leq \ D_t, \quad D_t \leq T_t$$

where O_h is the head's offset, and where D_t and T_t are the tail's deadline and period, respectively.

Example. Revisiting the example, we have the following execution-time constraints.

$$\begin{array}{rlll} O_s + e_s \leq D_s, & O_s \geq 0, & D_s \leq T_s, & \text{sampler task} \\ O_i + e_i \leq D_i, & O_i \geq 0, & D_i \leq T_i, & i = \{4, 6\} \\ O_s + e_s + e_i \leq D_i, & D_i \leq T_i & & i = \{1, 2, 3\} \\ O_s + e_s + e_2 + e_5 \leq D_5, & D_5 \leq T_5 & & \end{array}$$

This completes the set of task-wise constraints $\mathcal{C}$ imposed on our ATG. Thus far we have shown only one part of the problem – how $\mathcal{C}$ can derived from the end-to-end constraints. The end-to-end requirements will be maintained during runtime (1) if a solution to $\mathcal{C}$ is found, and (2) if the scheduler dispatches the tasks according to the solution's periods, offsets and deadlines. Since there are many existing schedulers that can handle problem (2), we now turn our attention to problem (1).

4 Step 2: Constraint Solver

The constraint solver generates instantiations for the periods, deadlines and offsets. In doing so, it addresses the notion of feasibility by using objective functions which (1) minimize the overall system utilization; and (2) maximize the window of execution for each task. Unfortunately, the non-linearities in the optimization criteria – as well as the harmonicity assumptions – lead to a very complex search problem.

We present a solution which decomposes the problem into relatively tractable parts. Our decomposition is motivated by the fact that the non-linear constraints are confined to the period variables, and do not involve deadlines or offsets. This suggests a straightforward approach, which is presented in Figure 8.

1. The entire constraint set $\mathcal{C}$ is projected onto its subspace $\hat{\mathcal{C}}$, constraining only the T_i's.
2. The constraint set $\hat{\mathcal{C}}$ is optimized for minimum utilization.
3. Since we now have values for the T_i's, we can instantiate them in the original constraint set $\mathcal{C}$. This forms a new, reduced set of constraints $\bar{\mathcal{C}}$, all of whose functions are affine in the O_i's and D_i's. Hence solutions can be found via linear optimization.

The back-edge in Figure 8 refers to the case where the nonlinear optimizer finds values for the T_i's, but no corresponding solution exists for the O_i's and D_i's. Hence, a new instantiation for the periods must be obtained – a process that continues until either a solution is found, or all possible values for the T_i's are exhausted.

4.1 Elimination of Offset and Deadline Variables

We use an extension of Fourier variable elimination [6] to simplify our system of constraints. Intuitively, this step may be viewed as the projection of an n dimensional polytope (described by the constraints) onto its lower-dimensional shadow.

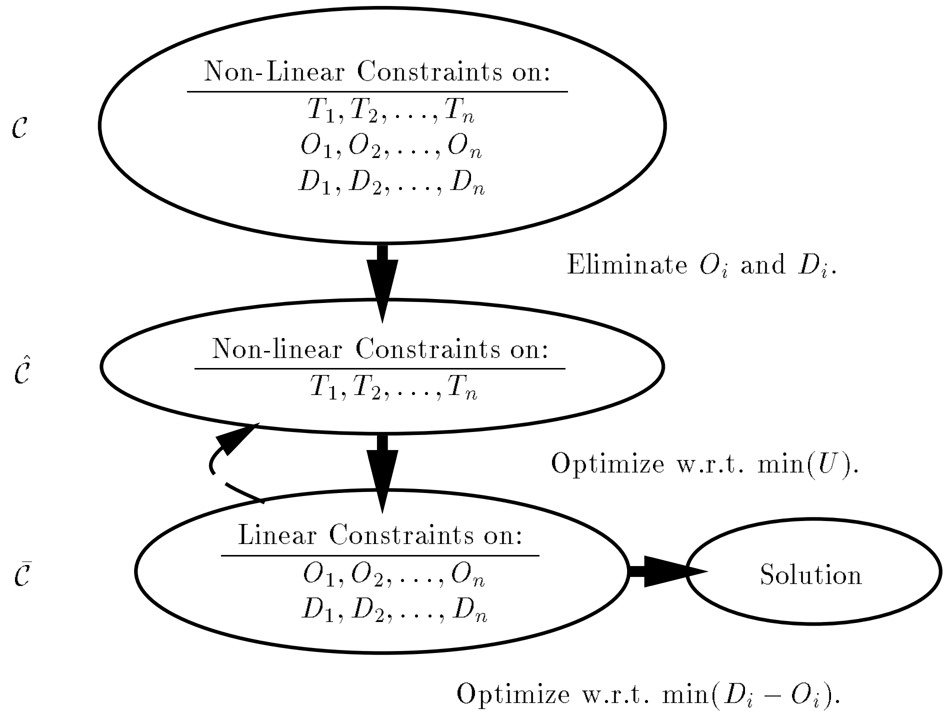


Figure 8: Top level algorithm to obtain task characteristics.

In our case, the n-dimensional polytope is the object described by the initial constraint set $\mathcal{C}$, and the shadow is the subspace $\hat{\mathcal{C}}$, in which only the T_i's are free. The shadow is derived by eliminating one offset (or deadline) variable at a time, until only period variables remain. At each stage the new set of constraints is checked for inconsistencies (e.g., $0 > 5$). Such a situation means that the original system was over-specified – and the method terminates with failure.

The technique can best be illustrated by a small example. Consider the following two inequalities on $W_4 = D_4 - O_4$:

$$W_4 \geq T_4 + 18 \qquad W_4 \leq 31 - T_4$$

Each constraint defines a line; when W_4 and T_4 are restricted to nonzero solutions, the result is a 2-dimensional polygon. Eliminating the variable W_4 is simple, and is carried out as follows:

$$\begin{aligned} & T_4 + 18 \leq W_4, \quad W_4 \leq 31 - T_4 \\ \Rightarrow \quad & T_4 + 18 \leq 31 - T_4 \\ \Rightarrow \quad & 2T_4 \leq 31 - 18 \\ \Rightarrow \quad & T_4 \leq 6.5 \end{aligned}$$

Since we are searching for integral, nonzero solutions to T_4, any integer in $[0 \ldots 6]$ can be considered a candidate.

When there are multiple constraints on W_4 – perhaps involving many other variables – the same

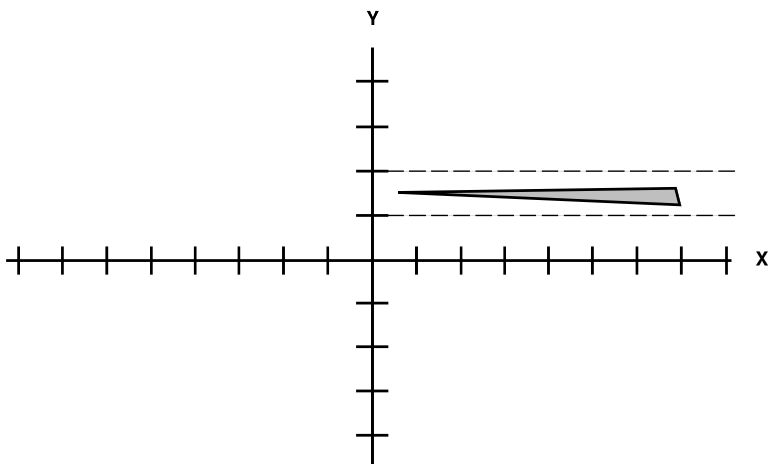


Figure 9: Variable elimination for integer solutions – A deviant case.

process is used. Every constraint "$W_4 \leq \ldots$" is combined with every other constraint "$W_4 \geq \ldots$," until W_4 has been eliminated. The correctness of the method follows simply from the polytope's convexity, i.e., if the original set of constraints has a solution, then the solution is preserved in the shadow.

Unfortunately, the opposite is not true; hence the the requirement for the back-edge in Figure 8. As we have stated, the refined constraint set $\hat{C}$ may possess a solution for the T_i's that do not correspond to any integral-valued O_i's and D_i's. This situation occasionally arises from our quest for integer solutions to the T_i's – which is essential in preserving our harmonicity assumptions.

For example, consider the triangle in Figure 9. The X-axis projection of the triangle has seven integer-solutions. On the other hand, none exist for Y, since all of the corresponding real-valued solutions are "trapped" between 1 and 2.

If, after obtaining a full set of T_i's, we are left without integer values for the O_i's and D_i's, we can resort to two possible alternatives:

1. Search for rational solutions to the offsets and deadlines, and reduce the clock-granularity accordingly, or

2. Try to find new values for the T_i's, which will hopefully lead to a full integer solution.

The Example Application – From $\mathcal{C}$ to $\hat{\mathcal{C}}$. We illustrate the effect of variable elimination on the example application presented earlier. The derived constraints impose lower and upper bounds on task periods, and are shown below. Also remaining are the original harmonicity constraints.

Linear Constraints	P_s	P_1	P_2	P_3	P_4	P_5	P_6
	$1 \leq T_s$	$7 \leq T_1$	$4 \leq T_2$	$4 \leq T_3$	$20 \leq T_4 \leq 29$	$7 \leq T_5$	$31 \leq T_6 \leq 39$
Harmonicity Constraints	$T_4 \vert T_1$, $T_1 \vert T_s$, $T_4 \vert T_2$, $T_2 \vert T_1$, $T_6 \vert T_5$, $T_5 \vert T_2$, $T_2 \vert T_s$, $T_6 \vert T_3$, $T_3 \vert T_s$						

Here the constraints on the output tasks (P_4 and P_6) stem from the separation constraints, which impose upper and lower bounds on the periods.

4.2 From $\hat{\mathcal{C}}$ to $\bar{\mathcal{C}}$: Deriving the Periods

Once the deadlines and offsets have been eliminated, we have a set of constraints involving only the task periods. The objective at this point is to obtain a feasible period assignment which (1) satisfies the derived linear equations; (2) satisfies the harmonicity assumptions; and (3) is subject to a realizable utilization, i.e., $U = \sum \frac{e_i}{T_i} \leq 1$.

As in the example above, the maximum separation constraints will typically mandate that the solution-space for each T_i be bounded from above. Thus we are faced with a decidable problem – albeit a complex one. In fact there are cases which will defeat all known algorithms. In such cases there is no alternative to traversing the entire Cartesian-space

$$[l_1, u_1] \times [l_2, u_2] \times \ldots [l_n, u_n]$$

where there are n tasks, and where each T_i may range within $[l_i, u_i]$. Fortunately the ATG's structure gives rise to a heuristics which can aggressively prune the search space. We call it *harmonic chain merging.*

Let $Pred(i)$ ($Succ(i)$) denote the set of tasks which are predecessors (successors) of task P_i, i.e., those tasks from (to) which there is a directed path to (from) P_i. Since the harmonicity relationship is transitive, we have that if $P_j \in Succ(P_i)$, it follows that $T_j | T_i$. This simple fact leads to the following observation: we do not have to solve for each T_i as if it is an arbitrary variable in an arbitrary function. Rather, we can combine chains of processes, and then solve for their base periods. This dramatically reduces the number of free variables.

For our purposes, this translates into the following rule:

> If a task P_i executes with period T_i, and if some $P_j \in Pred(P_i)$ has the property that $Succ(P_j) = \{P_i\}$, then P_j should also execute with period T_i.

In other words, we will never run a task faster than it needs to be run. In designs where the periods are ad-hoc artifacts, tuned to achieve the end-to-end constraints, such an approach would be highly unsafe. Here the rate constraints are *analytically derived* directly from the end-to-end requirements. *We know "how fast" a task needs to be run, and it makes no sense to run it faster.*

This allows us to simplify the ATG by merging nodes, and to limit the number of free variables in the problem. The method is summed up in the following steps:

(1) If $P_i \in Pred(P_j)$, then $T_j | T_i$ and consequently, $T_i \leq T_j$. The first pruning takes place by propagating this information to tighten the period bounds. Thus, for each task P_i, the bounds are tightened as follows:

$$l_i = \max\{l_k \mid P_k \in Pred(P_i)\}$$
$$u_i = \min\{u_k \mid P_k \in Succ(P_i)\}$$

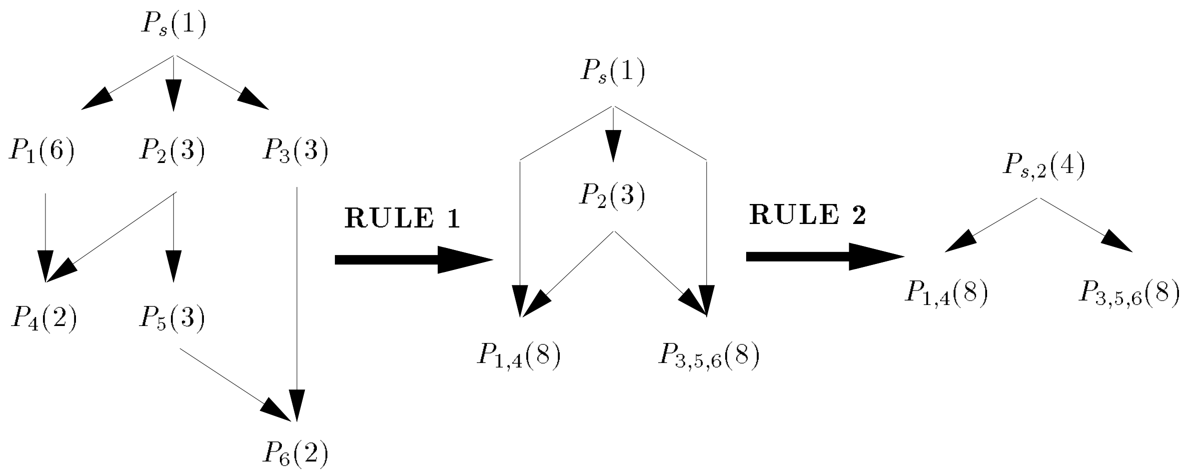


Figure 10: Task graph for harmonicity and its simplification.

(2) The second step in the algorithm is to simplify the task graph. Consider a task P_i, which has an outgoing edge $P_i \rightarrow P_j$. Suppose $u_i \geq u_j$. Then the maximum value of T_i is constrained only by harmonicity restrictions. The simplification is done by merging P_i and P_j, whenever it is safe to set $T_i = T_j$, i.e., the restricted solution space contains the optimal solution. The following two rules give the condition when it safe to perform this simplification.

Rule 1: If a vertex P_i has a single outgoing edge $P_i \rightarrow P_j$, then P_i is merged with P_j.

Rule 2: If $Succ(P_i) \subseteq (Succ(P_j) \cup \{P_j\})$ for some edge $P_i \rightarrow P_j$, then P_i is merged with P_j.

Consider the graph in Figure 10. The parenthesized numbers denote the costs of corresponding nodes. In the graph, the nodes P_3, P_5, and P_1 have a single outgoing edge. Using **Rule 1**, we merge P_3 and P_5 with P_6, and P_1 with P_4. In the simplified graph, $Succ(P_s) = \{P_4, P_6, P_2\}$ and $Succ(P_2) = \{P_4, P_6\}$. Thus, we can invoke **Rule 2** to merge P_s with P_2. Also, our three merged tasks have the following allowable ranges:

$$\begin{array}{lcl} P_{s,2} & : & \{T_{s,2} \mid 4 \leq T_{s,2} \leq 29\} \\ P_{1,4} & : & \{T_{1,4} \mid 20 \leq T_{1,4} \leq 29\} \\ P_{3,5,6} & : & \{T_{3,5,6} \mid 31 \leq T_{3,5,6} \leq 39\} \end{array}$$

This scheme manages to reduce our original seven tasks to three sets of tasks, where each set can be represented as a pseudo-task with its own period, and an execution time equal to the sum of its constituent tasks.

At this point we have reduced the structure of the ATG as much as possible, and we turn to examining the search process. But the size of the search space can still be enormous, even for a modest ATG. For example, 10 free period variables, each of which contains 10 possible values constitute a space of 10^{10} solutions. Fortunately, harmonicity requirements play a significant roll to reduce the search effort, since we need to look at only those period values that are integral multiples of certain base periods. In [8] we presented a graph-theoretic algorithm which is capable of finding a feasible solution relying on backward and forward traversal of a task graph. Here we sketch the idea behind the algorithm; interested readers should consult [8] for the technical details.

To sum up, the algorithm has the following properties.

(1) Period assignment is done in topological order. For a chain of tasks "$P_1 \rightarrow P_2 \rightarrow \cdots \rightarrow P_k$," each T_i must be an integral multiple of T_1, and thus T_i can be written as $a_i T_1$ for some $a_i \geq 1$. Whenever such a solution for T_i cannot be found, new periods for the immediate predecessors are determined.

(2) Whenever the system utilization approaches 100%, the current solution is rejected.

The algorithm can best be illustrated by our task graph in Figure 10: The T_i's are rewritten as below:

$$\begin{array}{lllll} P_{s,2} & : & T_{s,2} & & \\ P_{1,4} & : & T_{1,4} & = & a_1 T_{s,2} \\ P_{3,5,6} & : & T_{3,5,6} & = & a_2 T_{s,2} \end{array}$$

Now, feasible values are investigated for $T_{s,2}$, a_1 and a_2. First, $T_{s,2}$ is assigned its maximum allowable value 29, with leads to setting $a_1 = 1$. But with this assignment, no feasible value can be found for a_2. So a smaller value is tried for $T_{s,2}$, and so on. This process repeats until the algorithm terminates with a valuation of $T_{s,2} = 13$, $a_1 = 2$ and $a_2 = 3$ – which forms a feasible solution.

4.3 Deriving Offsets and Deadlines

Once the task periods are determined, we need to revisit the constraints to find a solution to the deadlines and offsets of the periods. Here, the residue of variable elimination allows us to select values in the reverse order in which they are eliminated. Suppose we performed elimination on the following variables, in order: $x_1, x_2, \ldots, x_n$. When x_i is eliminated, the remaining free variables are $[x_{i+1}, \ldots, x_n]$. Since $[x_{i+1}, \ldots, x_n]$ are already bound to values, the constraints immediately give a lower and an upper bound on x_i.

We use this fact in assigning offsets and deadlines to the tasks. As the variables are assigned values, each variable can be individually optimized. Recall that the feasibility of a task set requires that the task set never demand a utilization greater than 1 in any time interval. We use a greedy heuristic, which attempts to maximize the window of execution for each task. For tasks which do not have an offset, this is straightforward, and can be achieved by maximizing the deadline. For input/output tasks which have offsets, we also need to fix the position of the window on the time-line. We do this by minimizing the offset for input tasks, and maximizing the deadline for output tasks.

The order in which the variables are assigned is given by the following strategy: First, we assign the windows for each input task, followed by the windows for each output task. Then, we assign

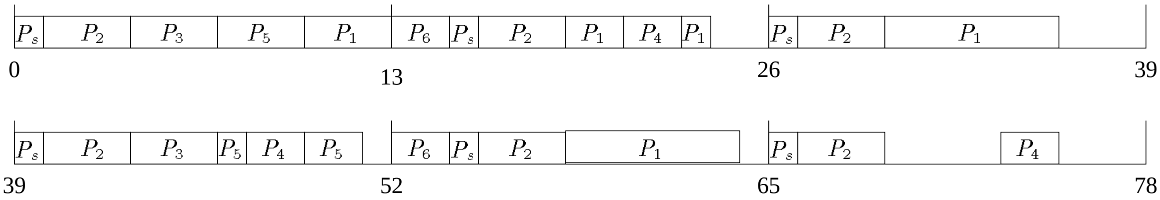


Figure 11: Feasible schedule for example application.

the offsets for each task followed by deadline for each output task. Finally, the deadlines for the remaining tasks are assigned in a reverse topological order of the task graph. Thus, an assignment ordering for the example application is given as $\{W_s, W_4, W_6, O_s, D_4, D_6, D_5, D_3, D_1, D_2\}$. The final parameters, derived as a result of this ordering, are shown below.

	P_s	P_1	P_2	P_3	P_4	P_5	P_6
Period	13	26	13	39	26	39	39
Offset	0	0	0	0	21	0	13
Deadline	3	21	13	13	26	13	15
Exec. Time	1	6	3	3	2	3	2

A feasible schedule for the task set is shown in Figure 11. We note that the feasible schedule can be generated using the fixed priority ordering $P_6, P_s, P_4, P_2, P_3, P_5, P_1$.

5 Step 3: Graph Transformation

When the constraint-solver fails, replicating part of a task graph may often prove useful in reducing the system's utilization. This benefit is realized by eliminating some of the tight harmonicity requirements, mainly by decoupling the tasks that possess common producers. As a result, the constraint derivation algorithm has more freedom in choosing looser periods for those tasks.

Recall the example application from Figure 5(B), and the constraints derived in Section 4. In the resulting system, the producer/consumer pair (P_2, P_5) has the largest period difference ($T_2 = 13$ and $T_5 = 39$). Note that the constraint solver mandated a tight period for P_2, due to the coupled harmonicity requirements $T_4|T_2$ and $T_5|T_2$. Thus, we choose to replicate the chain including P_2 from the sampler (P_s) to data object d_2. This decouples the data flow to Y_1 from that to Y_2. Figure 12 shows the result of the replication.

Running the constraint derivation algorithm again with the transformed graph in Figure 12, we obtain the following result. The transformed system has a utilization of 0.7215, which is significantly lower than that of the original task graph (0.8215).

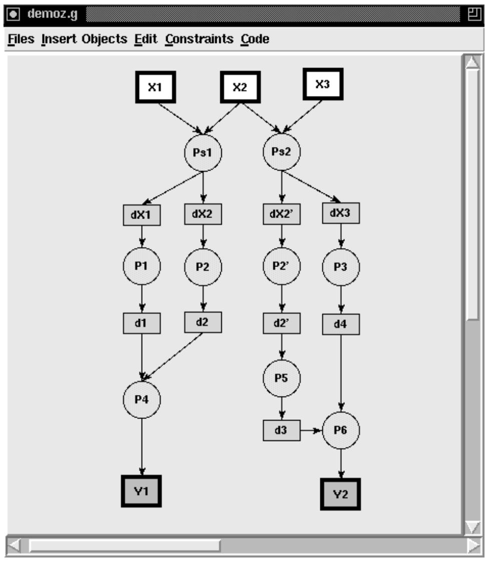


Figure 12: The replicated task graph.

	P_{s1}	P_1	P_2	P_4	P_{s2}	P'_2	P_3	P_5	P_6
Periods	29	29	29	29	39	39	39	39	39
Exec. Time	1	6	3	2	1	3	3	3	2

The subgraph replication technique begins with selecting a producer/consumer pair which requires replication. There exist two criteria in selecting a pair, depending on the desired goal. If the goal is reducing expected utilization, a producer/consumer pair with the maximum period difference is chosen first. On the other hand, if the goal is achieving feasibility, then we rely on the feedback from the constraint solver in determining the point of infeasibility.

After a producer/consumer pair is selected, the algorithm constructs a subgraph using a backward traversal of the task graph from the consumer. In order to avoid excessive replication, the traversal is terminated at the first confluence point. The resulting subgraph is then replicated and attached to the original graph.

The producer task in a replication may, in turn, be further specialized for the output it serves. For example, consider a task graph with two consumers P_{c1} and P_{c2} and a common producer P_p. If we replicate the producer, we have two independent producer/consumer pairs, namely (P_p, P_{c1}) and (P'_p, P_{c2}). Since P'_p only serves P_{c2}, we can eliminate all operations that only contribute to the output for P_{c1}. This is done by *dead code elimination*, a common compiler optimization. The same specialization is done for P_p.

6 Step 4: Buffer Allocation

Buffer allocation is the final step of our approach, and hence applied to the feasible task graph whose timing characteristics are completely derived. During this step, the compiler tool determines the buffer space required by each data object, and replaces its associated reads and writes with simple macros. The macros ensure that each consumer reads temporally correlated data from several data objects – even when *these* objects are produced at vastly different rates. The reads and writes are nonblocking and asynchronous, and hence we consider each buffer to have a "virtual sequence number."

Combining a set of correlated data at a given confluence point appears to be a nontrivial venture. After all, (1) producers and the consumers may be running at different rates; and (2) the flow delays from a common sampler to the distinct producers may also be different. However, due to the harmonicity assumption the solution strategy is quite simple. Given that there are sufficient buffers for a data object, the following rule is used:

> "Whenever a consumer reads from a channel, it uses the *first* item that was generated within *its* current period."

For example, let P_p be a producer of a data object d, let $P_{c_1}, \ldots, P_{c_n}$ be the consumers that read d. Then the communication mechanism is realized by the following techniques (where $L = LCM_{1 \leq i \leq n}(T_{c_i})$ is the least common multiple of the periods):

(1) The data object d is implemented with $s = L/T_p$ buffers.

(2) The producer P_p circularly writes into each buffer, one at a time.

(3) The consumer P_{c_i} reads circularly from slots $(0, T_{c_i}/T_p, \ldots, m \cdot T_{c_i}/T_p)$ where $m = L/T_{c_i} - 1$.

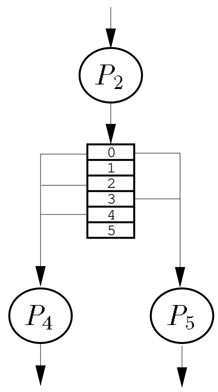
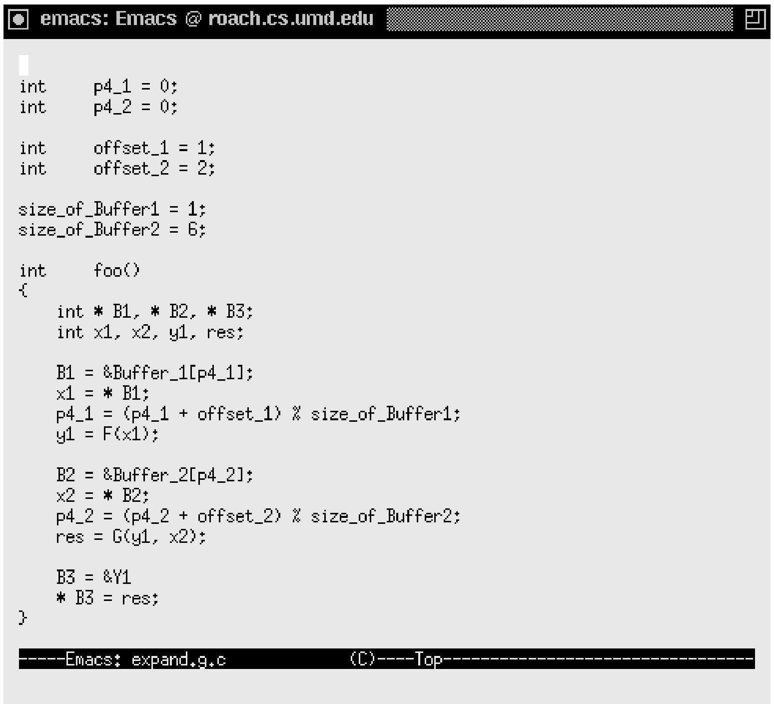


Figure 13: A task graph with buffers.

Consider three tasks P_2, P_4 and P_5 in our example, before we performed graph replication. The two consumer tasks P_4 and P_5 run with periods 26 and 39, respectively, while the producer P_2 runs with period 13. Thus, the data object requires a 6 place buffer ($6 = LCM(26,39)/13$), and P_4




```
int     p4_1 = 0;
int     p4_2 = 0;

int     offset_1 = 1;
int     offset_2 = 2;

size_of_Buffer1 = 1;
size_of_Buffer2 = 6;

int     foo()
{
    int * B1, * B2, * B3;
    int x1, x2, y1, res;

    B1 = &Buffer_1[p4_1];
    x1 = * B1;
    p4_1 = (p4_1 + offset_1) % size_of_Buffer1;
    y1 = F(x1);

    B2 = &Buffer_2[p4_2];
    x2 = * B2;
    p4_2 = (p4_2 + offset_2) % size_of_Buffer2;
    res = G(y1, x2);

    B3 = &Y1
    * B3 = res;
}
```


Figure 14: Instantiated code with copy-in/copy-out channels and memory-mapped IO.

reads from slots (0, 2, 4) while P_5 reads from slots (0, 3). Figure 13 shows the relevant part of the task graph after the buffer allocation.

After the buffer allocation, the compiler tool expands each data object into a multiple place buffer, and replaces each read and write operations with macros that perform proper pointer updates. Figure 14 shows the results of the macro-expansion, after it is applied to P_4's code from Figure 2(B). Note that P_1, P_2 and P_4 run at periods of 26, 13 and 26, respectively.

7 The Prototype Implementation

The objectives of the DesignAssistant are as follows.

(1) To provide a rapid prototyping tool for real-time system designers, so they can quickly build a running system for various analyses and optimizations.

(2) To let developers easily pin-point bottlenecks in the system.

(3) To help developers transform faulty components in their systems.

(4) To provide traceability between the entity-relationships in the high-level system design, and their manifestation in the low-level module code.

To achieve the the fourth goal, the DesignAssistant runs all tests from the same user interface in which the system topology is designed. The result is toolkit driver, whose operations allow drawing the system structure, binding code modules to task nodes, solving the constraints, and producing a Makefile to generate the application.

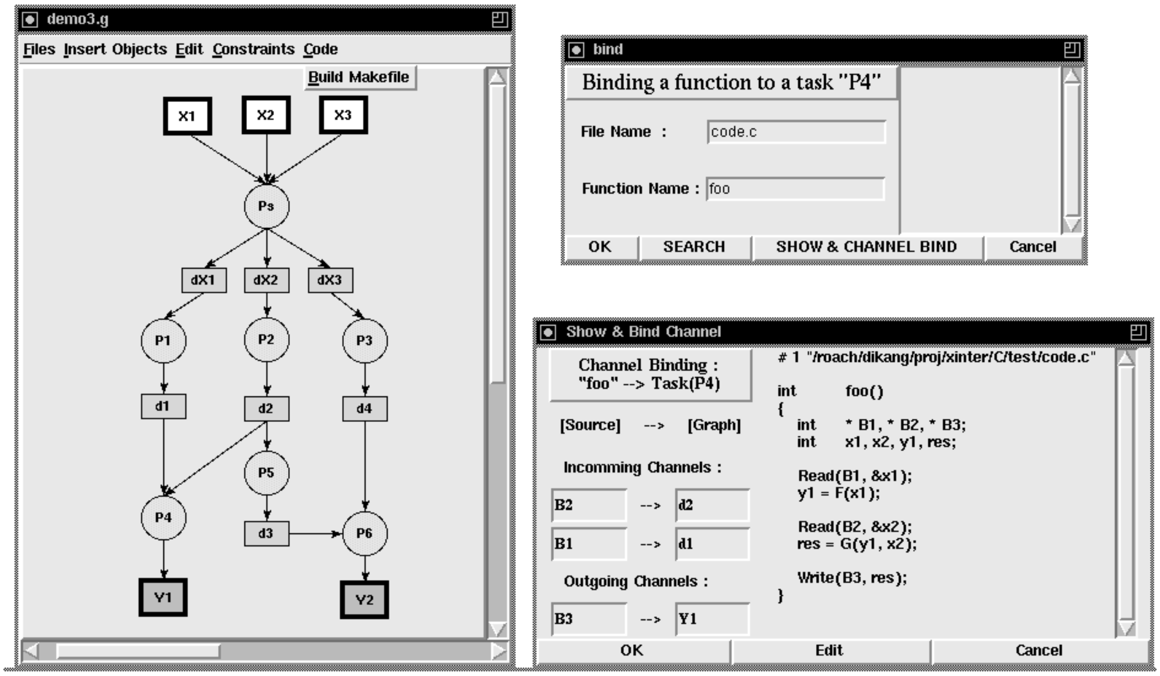


Figure 15: TimeWare/DesignAssistant Tool Screen.

7.1 Graphical User Interface

Figure 15 shows three of tool screens, all of which were implemented using Tcl/Tk toolkits [14]. The interactive graph editor (Figure 15(Left)) supports drawing and structuring the ATGs, and it allows hierarchical decomposition of ATGs based on black-box abstraction. That is, a sub-module node in an ATG can then be expanded into another ATG, with its inner structure drawn on a separate window. Figure 16 shows a hierarchical design equivalent to the "flat ATG" in Figure 15(Left). The sub-module's interfaces are drawn as small eclipses tagged by the connecting channels' names. For example, Figure 16(Right) shows that the buffer "dX1" is an external connection to the submodule "Sub_M1".

Each task node in an ATG must be associated with a piece of code which actually carries its computation. Since a single code module may be used for multiple task nodes in a number of distinct ATGs, it is necessary to support binding the abstract names in the ATG to associated names within the code modules. Task and channel binding are shown in the top and bottom right windows of Figure 15, respectively. In Figure 15, node "P4" is bound to the function "foo" in the file "code.c," and channels B1, B2, and B3 are bound to d1, d2, and Y1 respectively.

7.2 Constraint Solving

After accepting an ATG and associated attributes, our tool is ready to compute the task-specific parameters, using the algorithms we presented above. Computing the parameters consists of two parts: parsing and solving. Constraints accepted by the DesignAssistant are stored in a text file, as

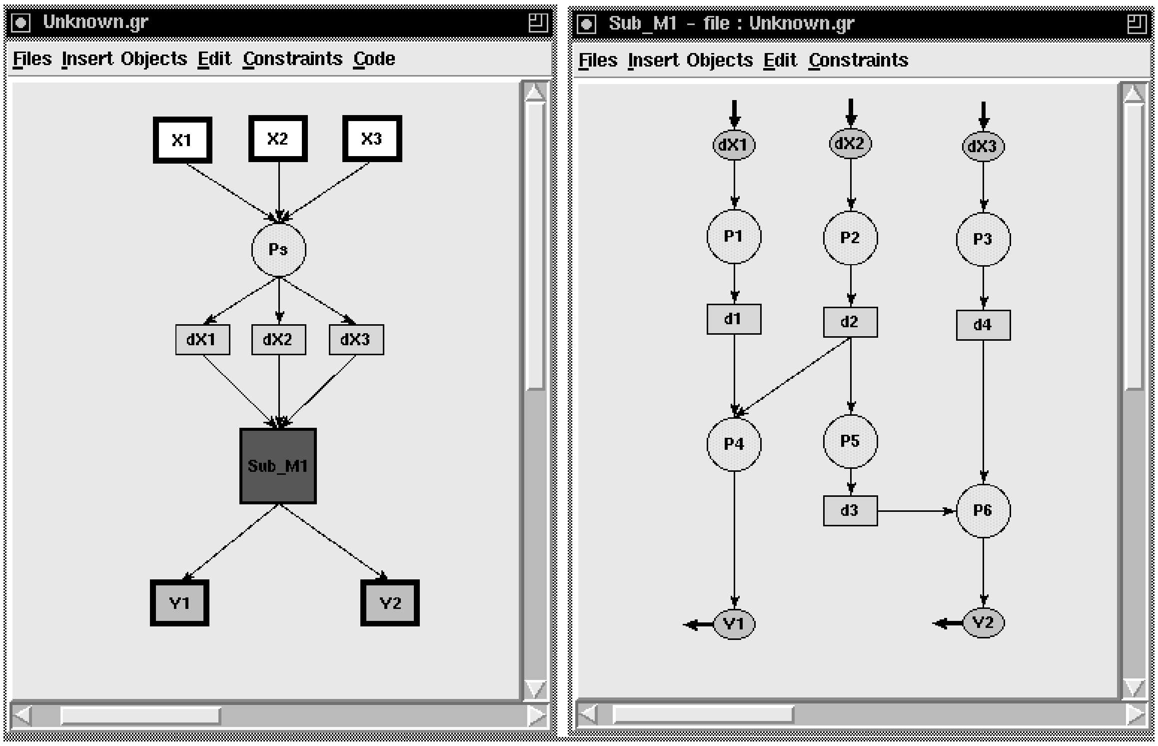


Figure 16: TimeWare/DesignAssistant Hierarchical Design Support.

shown in Figure 3. Clicking the `Solve` button invokes parsing the constraints file, and generating an intermediate form for the solver to use. The solver's job is then to derive the periods, offsets, and deadlines, and to determine the size of each channel. If no such solution exists, an error is reported, and restructuring of the ATG is required.

7.3 Code Generation

When the analysis ensures that the design is consistent, the tool will produce compilable code via a Makefile. Clicking `Build Makefile` in within the `Code` menu (Figure 15(Left)) results in the the following three actions.

(1) An initialization module for the application is created, and buffer allocation code is inserted within it.

(2) Buffer pointers are instantiated for each producer-consumer relationship, and the "**Read**" and "**Write**" operations are replaced by specialized access code.

(3) A Makefile is produced to compile executable applications.

7.4 Current Limitations and Future Extensions

Even within the scope of single-CPU applications, the implementation still possesses several limitations. First, nodes in an ATG can only be instantiated by source code and not, for example, a binary application which uses standard-input and output to communicate. The reason is obvious: our buffer allocation and instantiation is realized by source code translation.

Another limitation is that the Makefile, when executed, generates a monolithic compiled kernel which implements the given ATG.

But these two problems are induced by the properties of most existing runtime system, and are not inherent to our design methodology. The problem is simple: there exist many different kernel models, each of which possesses its own native input-output mechanisms. Nonetheless, we are currently investigating specific real-time operating systems, and associated threads packages, which should allow us to relax these limitations.

8 Conclusion

We have presented a four-step design methodology to help synthesize end-to-end requirements into full-blown real-time systems. Our framework can be used as long as the following ingredients are provided: (1) the entity-relationships, as specified by an asynchronous task graph abstraction; and (2) end-to-end constraints imposed on freshness, input correlation and allowable output separation. This model is sufficiently expressive to capture the temporal requirements – as well as the modular structure – of many interesting systems from the domains of avionics, robotics, control and multimedia computing.

However, the asynchronous, fully periodic model does have its limitations; for example, we cannot support high-level blocking primitives such as RPCs. On the other hand this deficit yields significant gains; e.g., handling streamed, tightly correlated data solely via the "virtual sequence numbers" afforded by the rate-assignments.

There is much work to be carried out. First, the constraint derivation algorithm can be extended to take full advantage of a wider spectrum of timing constraints, such as those encountered in input-driven, reactive systems. Also, we can harness finer-grained compiler transformations such as *program slicing* to help transform tasks into read-compute-write-compute phases, which will even further enhance schedulability. We have used this approach in a real-time compiler tool [7], and there is reason to believe that its use would be even more effective here.

We are also streamlining our search algorithm, by incorporating scheduling-specific decisions into the constraint solver. We believe that when used properly, such policy-specific strategies will help significantly in pruning the search space.

But the greatest challenge lies in extending the technique to distributed systems. The output and its inputs do not necessarily reside in the same processor in distributed systems, and there may be arbitrary numbers of network links from an input to the output. The characteristics of the network should be considered in conjunction with those constraints presented earlier on. Accordingly, constraints solving strategy should be changed to reflect the network characteristic as another constraints. Certainly a global optimization is impractical, since the search-space is much too large. Rather, we are taking a compositional approach – by finding approximate solutions for each node, and then refining each node's solution-space to accommodate the system's bound on network utilization.

Acknowledgements

The authors gratefully acknowledge Bill Pugh, Jeff Fischer, Ladan Gharai and Tefvik Bultan, whose valuable suggestions greatly enhanced the quality of this work.

References

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proc. of IEEE Symposium on Logic in Computer Science*, 1990.

[2] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In *Proceedings of IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, May 1991.

[3] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Data consistency in hard real-time systems. Technical Report YCS 203 (1993), Department of Computer Science, University of York, England, June 1993.

[4] G. Berry, S. Moisan, and J. Rigault. ESTEREL: Towards a synchronous and semantically sound high level language for real time applications. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 30–37. IEEE Computer Society Press, December 1983.

[5] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. In S. Son, editor, *Principles of Real-Time Systems*. Prentice Hall, 1994.

[6] G. Dantzig and B. Eaves. Fourier-Motzkin Elimination and its Dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[7] R. Gerber and S. Hong. Semantics-based compiler transformations for enhanced schedulability. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 232–242. IEEE Computer Society Press, December 1993.

[8] R. Gerber, S. Hong, and M. Saksena. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 192–203. IEEE Computer Society Press, December 1994. Also to appear in *IEEE Transactions on Software Engineering*.

[9] M. Harbour, M. Klein, and J. Lehoczky. Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority. In *Proceedings, IEEE Real-Time Systems Symposium*, pages 116–128, December 1991.

[10] F. Jahanian and A. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, September 1986.

[11] K. Jeffay. The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *ACM/SIGAPP Symposium on Applied Computing*, pages 796–804. ACM Press, February 1983.

[12] M. Klein, J. Lehoczky, and R. Rajkumar. Rate-monotonic analysis for real-time industrial computing. *IEEE Computer*, pages 24–33, January 1994.

[13] C. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[14] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[15] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Software Engineering*, 39:1175–1185, September 1990.

[16] K. Tindell. Using offset information to analyse static priority pre-emptively scheduled task sets. Technical Report YCS 182 (1992), Department of Computer Science, University of York, England, August 1992.

[17] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *The Journal of Real-Time Systems*, 6(2):133–152, March 1994.

[18] J. Xu and D. Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, March 1990.

[19] X. Yuan, M. Saksena, and A. Agrawala. A Decomposition Approach to Real-Time Scheduling. *Real-Time Systems*, 6(1), 1994.

[20] W. Zhao, K. Ramamritham, and J. Stankovic. Scheduling Tasks with Resource requirements in a Hard Real-Time System. *IEEE Transactions on Software Engineering*, SE-13(5):564–577, May 1987.