Copyright 1998 IEEE. Published in the Proceedings of Rtss'98, 1 December 1998 in Madrid, Spain. Personal use of this material is permitted. However, permission to reprint/ republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 732-562-3966.

Detailed Design of Avionics Control Software*

Ulf Nilsson, Siwert Streiffert, Anders Törne
Department of Computer and Information Science
Linköping University, S-581 83 Linköping, SWEDEN
{*ulfni, siwst, andto*}*@ida.liu.se*

Abstract

Avionics control systems are typical real-time systems – automatic control algorithms are realized as periodic computations where strict requirements on the timing can be derived by a control analysis. The requirements are effectively end-to-end timing requirements on the computation – from sensory input to actuator output – and have to be transformed into a detailed control software design. This analysis, together with a resource and performance analysis, is required early in the development process – neither hardware nor prototypes of the software can be assumed to exist – in order to estimate the cost of development and the cost of hardware to meet customer requirements. The paper investigates the applicability of a method (the period calibration method) to transform the end-to-end requirements of an avionics control software, into a design consisting of a set of periodic tasks with buffered data flows. The experience is discussed and shortcomings of the method are analysed.

1. Introduction

An *avionics control system* is a typical real-time system involving many difficult decisions regarding the balance between performance, flexibility, production costs, maintenance costs, engineering effort, and total system functionality. Several of these decisions are made very early in the design process or have to be approximated during a quotation process. Typical decisions are the control laws, the partitioning of software and hardware functionality, the selection of basic hardware components and the architectural decisions regarding the software organization.

During *control design* the operational requirements on the total system are transformed into a set of derived functional (e.g. control laws) and non-functional (e.g. end-to-end timing) requirements. During *preliminary software design* a functional software design is made and a hardware platform is selected. Then the functional design and the timing requirements are mapped to the computational resources and the communication mechanisms are decided. During the *detailed software design* the functionality (possibly refined) for each computational resource is transformed into a set of periodic tasks, with timing attributes determined from the timing requirements. Finally an implementation specification for the software can be generated (*software module design*). The software/ hardware functional partitioning problem is not considered in this work – all control functionality is assumed to be implemented in software.

The goal of each design phase is to find a feasible[1] design while optimizing with respect to *non-strict* requirements, like minimizing life cycle costs (for obvious reasons), minimizing the constraints on later design phases (to make it easier to find a design), and minimizing the resource requirements (to simplify future extensions).

The design problem can be decomposed into:

- generating an "optimal" proposal for a feasible design – the synthesis problem. If the proposal satisfies the strict requirements by construction, then no analysis is needed.
- analyzing if the proposed design is feasible, i.e., if it satisfies the strict requirements – the analysis problem.

Industrially the synthesis and analysis are done iteratively until a satisfactory design is found. The design often has to be approximated during a quotation process. Neither hardware nor prototypes of the software can therefore be assumed to exist. This means, for example, that required attributes, like the *WCET*s, for a correct analysis of timing requirements have to be estimated.

If a design is found infeasible or problematic during control, preliminary or detailed design, the problem will be reiterated in a preceding phase, and a change of requirements will be made. Early design errors which

* This work was done on commission from Saab Dynamics AB.

1. A feasible design is one that satisfies the strict requirements – i.e., requirements that are either satisfied or not.

cause problems in later phases will therefore be costly when the complexity of the design increases. To increase the quality of early designs it is therefore crucial to develop tools and methods to support the generation of the design and the verification of the requirements.

Design methods for real-time software normally only provide a notation for the model of the design, a description of inputs/outputs for each phase in the development process, and general descriptions of how to perform the phases. The tools do often lack support for functional and/or timing verification (analysis). Even more rare is support for generating the designs (synthesis) given input requirements.

Typical design methods are HRT-HOOD (cf. [5, 6]) and the MARS method (cf. [11, 14]). HRT-HOOD requires a method to estimate (or determine) *WCET*s and a decision on scheduling strategy. HRT-HOOD is more general than MARS, since MARS has the hardware architecture and programming environment defined together with the method. On the other hand it is easier to incorporate synthesis and analysis tools (like a WCET-analyzer) with MARS.

The long term intention of our research is to investigate, evaluate and improve industrially usable computerized support for real-time system design, that complements the used design method. This paper describes a feasibility study of a method (the period calibration method [8]) and tool to support synthesis during the detailed design phase. The objectives (related to the presentation in this paper) were:

- to evaluate the selected method.
- to investigate tool support for the method.
- analyze the benefits and drawbacks.

We also assume that the control analysis has resulted in functional and timing requirements (strict requirements) on the software/hardware realization and that the preliminary design has been made. It is furthermore assumed that the algorithmic realization of the control laws is straightforward during modular design and therefore does not strongly interfere with the earlier phases.

The work presented here was made in cooperation with Swedish aerospace industry, as a continuation on an earlier project which investigated early estimation of worst case execution times, methods to develop control software, and ways to do schedulability analysis (cf. [4]).

This feasibility study first briefly presents the period calibration method. In Section 3 the actual case is presented. The tool used in the case study and the resulting implementation are presented in Sections 4 and 5. Finally we analyze benefits and drawbacks of the method.

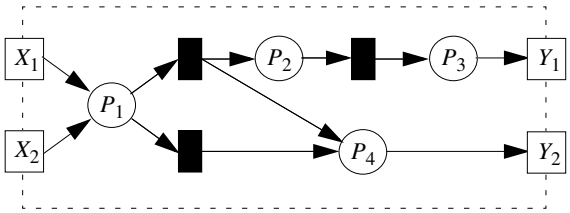


Figure 1: An asynchronous task graph showing the subtasks and the communication buffers (in black).

2. The Synthesis Method

In this section we give a brief account of the synthesis method used – for a detailed account see Gerber, Hong and Saksena [8]. In this paper we refer to the method as the *GHS-method* after the authors of the main report, since the feasibility study investigated the method presented there and not later extensions or modifications. Later work of the authors refer to the general method as the *period calibration method*.

The synthesis method results in a detailed design (an Asynchronous Task Graph, ATG) composed of a number of interacting and communicating *tasks*, ultimately receiving data from external inputs and delivering data to external outputs, as illustrated in Figure 1.

The GHS-method assumes that end-to-end and internal timing requirements can be quantitatively described by means of algebraic equations and inequalities relating unknown variables. The unknown variables correspond to attributes of internal computations while the end-to-end requirements and worst case execution times are assumed to be known.

2.1. The Input of the Method

The method requires the following as input:

- A task graph describing the dataflow dependencies between the external inputs and outputs, and all intermediate tasks, similar to Figure 1.
- The *WCET* (denoted e_i) of each task.
- A set of end-to-end timing requirements on the system.

The end-to-end requirements

The end-to-end requirements are assumed to belong to one of the following three categories (see [8] for details):

- *Freshness requirement*s (or *propagation delays*), $F(Y \mid X)$, bound the time for data to flow through the system from input X to output Y.
- *Correlation requirements,* $C(Y \mid X_1,X_2)$, limit the difference in sampling time between several inputs used to produce the same output.

- *Separation requirements* give the *lower, l*(*Y*), and *upper*, *u*(*Y*), time bound between consecutive values output on *Y.*

2.2. The Output of the Method

The method gives the following as output:

- A set of tasks with *offsets* (O_i), *deadlines* (D_i) and *periods* (T_i). The tasks include those given as input, plus possibly some new or transformed tasks;
- A set of data buffers with determined parameters for communication between tasks;
- An Asynchronous Task Graph (ATG) connecting tasks and buffers, as illustrated in Figure 1;
- A partial order of *priorities* to guarantee precedence constraints for internal tasks.

The method may also fail in producing these outputs.

If all tasks can be scheduled according to their attributes, then all of the end-to-end requirements will be met. Furthermore, the processor utilization is guaranteed to be below a maximum value (as discussed in the next section).

2.3. The Transformation

The transformation of the GHS-method from input to output deals with two main issues: creation of the ATG and derivation of task attributes. The structure of the ATG produced by the GHS-method is generally the same as the task graph provided as input, but extended by additional sampling tasks. If the attainable utilization proves to be too high, further restructuring of the graph may be necessary. A method named *task replication* that duplicates critical paths of the graph is described in [8]. This may relax the constraints on the solution and may facilitate finding a solution. Task replication was not examined in this case study, since the overall level of utilization was not a subject of concern.

In order to satisfy the *correlation* requirements one or more *sampling tasks* are added to the graph. The freshness constraints for inputs used to compute the same output, cannot differ more than the correlation for the same inputs. Thus the freshness constraints are trimmed to prevent this. The approach in the GHS-method is to actually make all such correlated freshness constraints equal.

The technique for dealing with freshness and separation requirements is to derive – from the given end-to-end requirements and the structure of the ATG – a set of constraints (algebraic equations with inequalities) on periods, deadlines, and offsets of each task. A constraint solver is then used to obtain explicit solutions to the constraints. The details may be found in [8], and are summarized below.

The solution to the algebraic inequalities is further constrained by the following constraints:

- harmonicity – the period of each task should be a multiple of the period of all preceding tasks in the graph;
- utilization – namely $U \leq U_{max}$.

The first requirement makes it possible to design the communication between consecutive tasks so that the tasks execute independently (but in sequence) using asynchronous data buffers. The second requirement is natural in avionics control systems. Both requirements impose non-linear constraints in periods – harmonization expressed as $T_i \cdot N = T_j$ ($N = 1, 2, \ldots$) and utilization U expressed as:

$$U = \sum \frac{e_i}{T_i}$$

By elimination of offset and deadline variables (using Fourier-Motzkin elimination [7, 8]), a (still non-linear) set of inequalities only in periods is obtained which extends the solution space. A period solution to the original set is also a solution to the eliminated set, but not the opposite. By use of various heuristics the solution space for the periods can be restricted. To select one of the remaining solutions the selection criterion may for example be to choose the solution to the reduced problem that minimizes the utilization.

The offsets and deadlines are then reintroduced and instantiated through a set of optimization criteria aiming to maximize the probability of schedulability.

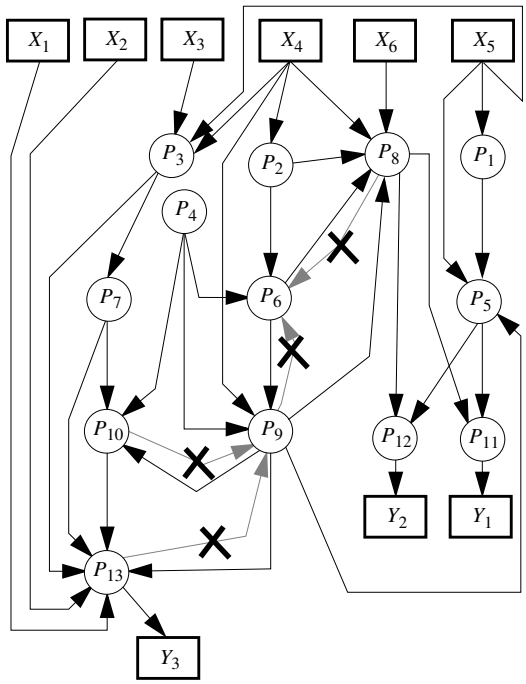
3. The Case Study

A full account of the intermediary results achieved during the application of the method cannot be given here due to space restrictions.

The example was provided by Swedish aerospace industry as a typical example of an avionics control system. The example was specified in a form adapted to the application of the GHS method – a computation graph, specification of in- and out-signals, maximal computational delays, correlations between insignals, and output separation requirements, and worst case execution times.

Some information was not available for confidentiality reasons, and was therefore invented (by industry) with the restriction that data *could* correspond to an actual design case. This issue will be further discussed later.

The computation graph is given in Figure 4, together with the end-to-end timing requirements. The computation graph differs from the graph used in the GHS-method in that it contains cyclic dependencies. For a direct application of the GHS-method these must be eliminated. No specific information was used to do this, but the choice was arbitrary. This issue will be further discussed below.



WCET

$e_1 = 2$ $e_8 = 2$
$e_2 = 2$ $e_9 = 2$
$e_3 = 9$ $e_{10} = 9$
$e_4 = 2$ $e_{11} = 2$
$e_5 = 2$ $e_{12} = 2$
$e_6 = 2$ $e_{13} = 2$
$e_7 = 9$

CORRELATION

$C(Y_1|X_4,X_5,X_6) = 10$
$C(Y_2|X_4,X_5,X_6) = 10$
$C(Y_3|X_1,X_2,X_3) = 10$

OUTPUT SEPARATION

$l(Y_1) = 90$ $u(Y_1) = 110$
$l(Y_2) = 90$ $u(Y_2) = 110$
$l(Y_3) = 180$ $u(Y_3) = 220$

FRESHNESS

$F(Y_1|X_4) = 50$
$F(Y_1|X_5) = 20$
$F(Y_1|X_6) = 50$
$F(Y_2|X_4) = 50$
$F(Y_2|X_5) = 20$
$F(Y_2|X_6) = 50$
$F(Y_3|X_1) = 50$
$F(Y_3|X_2) = 50$
$F(Y_3|X_3) = 50$

Figure 2: Computation graph for an avionics control system and related end-to-end timing requirements. Deleted dependencies are crossed out. (All timings are in milli-seconds.)

Another minor difference is the existence of a computation that lacks insignal (P_4). The reason for this is that this computation depends only on the global clock. The GHS-method can handle this by the constraints on period via harmonicity and utilization constraints.

All in- and out-signals were continuous.

3.1. Application of the Method

Gerber et al [8] propose a series of actions to perform the actual transformation from end-to-end constraints (input) to task-specific parameters (output). These actions where executed for the case study and are described below:

- First two sampling-tasks P_{s1} and P_{s2} are introduced that sample from inputs $\{X_1, X_2, X_3\}$ and $\{X_4, X_5, X_6\}$ respectively, and secondly, the freshness constraints $F(Y_1|X_i)$ and $F(Y_2|X_i)$ are trimmed (from 50 ms to 20 ms) as described in Section 2.3.
- Then the task graph is used to set up a number of constraints involving the *period* (T_i), *offset* (O_i) and *deadline* (D_i) of each task P_i. Each chain imposes *harmonicity constraints, e.g.,* $T_3 = N \cdot T_{s1}$, on the periods. The *correlation constraints* relate offsets and deadlines of the sampling tasks, e.g., $D_{s1} - O_{s1} \le 10$. The *freshness constraints* restrict the earliest sampling to the latest output time, e.g., $D_{13} - O_{s1} \le 50$. Note that not all computation chains had a defined freshness constraint – X_4 to Y_3 and X_5 to Y_3 were not specified. The precedence requirements are realized by *precedence constraints* only for input and output tasks ($D_{s1} \le D_3 - e_3$, $D_3 \le O_{13}$). Intermediary tasks all have $O_i = 0$ and are given a *priority ordering* so that the precedence requirements are satisfied. The output *separation constraints* have to consider that output can occur at any time between offset and deadline for an output task, e.g., $(T_{13} + D_{13}) - O_{13} \le 220$ and $(T_{13} - D_{13}) + O_{13} \ge 180$. The *execution time constraints* guarantee that each task gets a large enough window of execution, e.g., for input/output tasks

  $O_{s1} + e_{s1} \le D_{s1}$ $O_{s1} \ge 0$ $D_{s1} \le T_{s1}$
  $O_{13} + e_{13} \le D_{13}$ $O_{13} \ge 0$ $D_{13} \le T_{13}$

  and for intermediary tasks

  $O_{s1} + e_{s1} + e_3 \le D_3$ $D_3 \le T_3$

  The resulting number of equations and inequalities was approximately 100.
- Next the constraints are transformed by Fourier-Motzkin elimination of offsets and deadlines, resulting in equations in periods and harmonicity variables only. The harmonicity constraints and the utilization constraint are added to the constraint set (Figure 3).
- Then the *T*:s are resolved so that they satisfy the resulting system of inequalities. In [8] this task is supported in part by a set of heuristics. If the designer chooses, say, a maximum allowable utilization of 0.7, the heu-

ristics will delete *some* points in *T*-space that violate this assumption, but not necessarily (and not likely) *all* such points. Likewise the heuristics will remove *some* solutions that violate the harmonic constraints.

Applying the heuristics does not necessarily result in a unique value for each T_i, but rather a reduced solution space. It is up to the designer to select among these. The selected point in *T*-space should be checked to satisfy the original set of constraints, the harmonicity constraints and to imply a low enough utilization. In the current study we applied the heuristics manually. When the tool, described in Chapter 4, was applied, a depth first search strategy was used, without applying the solution space reduction heuristics. The same periods were achieved.

- Next the *O*:s and *D*:s and their corresponding constraints are reintroduced and an effort is made to maximize schedulability. This is done by a greedy heuristic that tries to maximize the execution window of every task by minimizing *O*:s and maximizing *D*:s as they are reintroduced in a certain order:

  First, the execution window (*D*–*O*) of all input tasks are maximized.

  Next, the execution window of all output tasks are maximized.

  Then the offset of all input tasks are minimized.

  Next the deadline of all output tasks are maximized.

  Finally, the deadlines of the intermediate tasks, sorted in topological order, are maximized.

$$T_{s1} \geq 1 \quad T_{s2} \geq 1 \quad T_1 \geq 3 \quad T_2 \geq 3 \quad T_3 \geq 10 \quad T_4 \geq 0$$
$$T_5 \geq 9 \quad T_6 \geq 0 \quad T_7 \geq 19 \quad T_8 \geq 9 \quad T_9 \geq 7 \quad T_{10} \geq 28$$
$$92 \leq T_{11} \leq 108 \quad 92 \leq T_{12} \leq 108 \quad 182 \leq T_{13} \leq 218$$

$$\frac{e_{s1}}{T_{s1}} + \frac{e_{s2}}{T_{s2}} + \frac{e_1}{T_1} + \frac{e_2}{T_2} + \frac{e_3}{T_3} + \ldots + \frac{e_{13}}{T_{13}} \leq U_{max}$$

$$T_3 = n_1 T_{s1} \quad T_{13} = n_2 T_{s1} \quad T_1 = n_3 T_{s2} \quad T_2 = n_4 T_{s2}$$
$$T_3 = n_5 T_{s2} \quad T_5 = n_6 T_{s2} \quad T_8 = n_7 T_{s2} \quad T_9 = n_8 T_{s2}$$
$$T_5 = n_9 T_1 \quad T_6 = n_{10} T_2 \quad T_8 = n_{11} T_2 \quad T_7 = n_{12} T_3$$
$$T_{13} = n_{13} T_3 \quad T_{11} = n_{14} T_5 \quad T_{12} = n_{15} T_5 \quad T_8 = n_{16} T_6$$
$$T_9 = n_{17} T_6 \quad T_{10} = n_{18} T_7 \quad T_{13} = n_{19} T_7 \quad T_{11} = n_{20} T_8$$
$$T_{12} = n_{21} T_8 \quad T_5 = n_{22} T_9 \quad T_8 = n_{23} T_9 \quad T_{10} = n_{24} T_9$$
$$T_{13} = n_{25} T_9 \quad T_{13} = n_{26} T_{10} \quad n_i \in \{1, 2, 3, \ldots\}$$

Figure 3: The complete set of constraints after Fourier-Motzkin elimination

There is no guarantee that the solution actually maximizes schedulability.

The constraints were set up and solved by means of a CLP-tool (see Chapter 4.). The initial (non-schedulable) solution together with a final solution are presented in Table 1.

- The design can now be analyzed w.r.t. schedulability. Note that utilization based schedulability analysis cannot be used since synchronization between tasks makes them dependent. Some problems were experienced at this stage so that the original problem had to be reformulated, see next section. The final solution using fixed priority scheduling is shown in Figure 4.
- If the derived set of parameters results in a schedulable set of tasks, asynchronous buffers can be allocated.

Table 1 : The solution using the GHS-method. Values in parentheses are derived using additional constraints resulting in a schedulable solution (with priorities).

P_i	e_i	T_i	O_i	D_i	Pri_i
*s*1	1	216	0	10	2
*s*2	1	108	0	10	1
1	2	108	0	108	4
2	2	108	0	108	3
3	9	216	0	46 (48)	12
4	2	108	0	108	5
5	2	108	0	18 (16)	9
6	2	108	0	108	6
7	9	216	0	46 (48)	13
8	2	108	0	18 (16)	8
9	2	108	0	46 (48)	7
10	9	216	0	46 (48)	14
11	2	108	18 (16)	20 (18)	10
12	2	108	18	20	11
13	2	216	46 (48)	50	15

3.2. Schedulability

Even if the average utilization was low (0.315), it is obvious from Table 1 that task 11 and 12 compete for the same window of execution in the initial solution. The utilization is at least 200 per cent in that window. The recommended approach given by the GHS-method in this situation is to transform the task graph, as described in Section 2.3. However, it is not obvious how this would solve the problem in our case. Instead a new inequality is added that guarantees that P_{11} executes before P_{12}, e.g., $D_{11} \leq O_{12}$. Furthermore, the sum of the *WCET*s of task 10 and its predecessors is greater (47 ms) than the deadline of task 10 (46 ms). Two additional constraints were there-

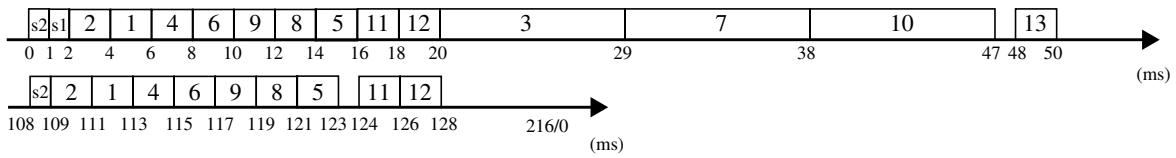


Figure 4: Timing diagram for the scheduled tasks according to the solution of the example.

fore added to set the offset and deadline of task 13; $O_{13} = 48$ and $D_{13} = 50$. It can be shown that the solution to the more constrained problem (derived using the same method as originally) is schedulable, given a certain order of priorities. (See Figure 4 and Table 1.)

3.3. Asynchronous Buffer Design

The buffers decouple the tasks and guarantee that producers and consumers of data never access the same location at the same time. As a result, locking is not required and tasks are never blocked by lower priority tasks. The design of the buffers presented no problem, and was straightforward using the GHS-method.

4. The Tool

The synthesis of *offsets*, *deadlines* and *periods* was supported by a small *constraint logic program* (see e.g. Marriott and Stuckey [13] or Jaffar and Maher [9]) implemented in SICStus Prolog [20]. Constraint logic programming (CLP) is a family of programming languages that combine logic programming with special-purpose algorithms for solving equations and inequations over various algebraic domains. In this particular project we used CLP(Q) – logic programming with linear constraints over the *rational* numbers. The combination of a general purpose programming language (with built-in search) and constraint solving provides a flexible and powerful tool for solving this type of problems – the prototype implementation consisted of less than 300 lines of code and was written in less than a day. Running the program on the example task graph (plus the sampling tasks that were added manually) took approximately 2.5 seconds on a 200MHz Pentium Pro running Linux.

The input to the program – the asynchronous task graph and the end-to-end requirements – was specified by means of a logic program; input tasks, output tasks and the execution time of all tasks were specified as follows:

```
task(s1, 1).     input(s1).     output(p11).
task(s2, 1).     input(s2).     output(p12).
task(p1, 2).     input(p4).     output(p13).
    ...
task(p13, 2).
```

The internal structure of the ATG was specified by relating the tasks to the list all of the immediate predecessors in the graph.

```
pred(p1, [ s2 ]).
pred(p2, [ s2 ]).
pred(p3, [ s1, s2 ]).
    ...
pred(p12, [ p5, p8 ]).
pred(p13, [ s1, p3, p7, p9, p10 ]).
```

Finally the external requirements were specified:

```
freshness(p13, s1, 50).    separation(p11, 90, 110).
freshness(p12, s2, 20).    separation(p12, 90, 110).
freshness(p11, s2, 20).    separation(p13,180, 220).

correlation(s1, 10).
correlation(s2, 10).
```

The main procedure of the program was responsible for *setting up* the constraints, and then *solving* them in accordance with the GHS-method:

```
main(Tasks, Utilization) :-                          (1)
    create_variables(Tasks),                         (2)
    setup_correlation(Tasks),                        (3)
    setup_freshness(Tasks),                          (4)
    setup_separation(Tasks),                         (5)
    setup_execution(Tasks),                          (6)
    setup_harmonicity(Tasks,Harm),                   (7)
    solve_constraints(Tasks,Harm,Utilization).       (8)
```

The predicate *create_variables*(*Tasks*) in line 1 was responsible for creating and initializing the unknowns (*offset*, *deadline* and *period*) for each task in the task graph. The variables were kept in *Tasks* (as an association list).

In lines 2–7 the different types of constraints (correlation, freshness etc.) were set up. For instance, the code for setting up the correlation constraints looked as follows:

```
setup_correlation(Tasks) :-
    collect_correlation(List),
    setup_corr_loop(List, Tasks).
```

First we collect the list of all tasks that need to be correlated and the corresponding correlation value. Then we traverse this list and set up the correlation constraint that restricts the window of execution of tasks that sample several inputs:

```
setup_corr_loop([ ], Tasks).
setup_corr_loop([ c(Task, Corr) | Rest ], Tasks) :-
    get_task(Task, Tasks, [Offset,Deadline,Period]),
    { Deadline – Offset =< Corr },
    setup_corr_loop(Rest, Tasks).
```

For each pair of task-correlation in the list we retrieve the associated offset and deadline (using *get_task*/3) and set up the constraint *Deadline – Offset =< Corr.*

The last step – before solving the constraints – was to generate and set up the constraints containing the integer-valued variables that govern the harmonicity of the system (line 7). The process of setting up the constraints was completely deterministic and driven by the structure of the ATG. Only the last step – actually finding a solution to the constraints (line 8) – is non-deterministic. This is also the most difficult part of the process computationally; it involved solving several non-linear constraints. Although CLP(Q) accepts non-linear constraints there is no support for solving or even checking the satisfiability of non-linear constraints. However, by exploiting the non-determinism of logic programming it was possible to reduce the problem to a linear one by systematically labeling the harmonicity variables using a depth first search. Solving the constraints was then easy.

5. The Implementation

An investigation of how to implement the detailed design in Ada 95 was made. Ada 95 [1] supports monotone time with a resolution of 1 ms and monotonicity of 50 years to safely implement cyclic tasks and offsets. It also supports static priorities and pre-emptive scheduling. The implementation framework contains three packages – one for the generic task scheduling administration, one for the application code and one for initialization. Exception handlers can be associated with missed deadlines

6. Discussion

In this section we present conclusions from the project and discuss ideas for extensions. Finally we present some related work.

6.1. Evaluation of the Method

Problem Specification

The example problem was specified by industry in a form prepared with the GHS-method in mind. The information was partly available, but sometimes in alternative forms. Some of the input data was therefore estimated, like *WCET* for the different tasks. The initial task graph contained several feedbacks which had to be eliminated. For confidentiality reasons, the original specification is not available to the authors, and no statement can presently be made about how well the method *really* fits industrial practice. However, the problem formulation, the problems encountered during the application of the method and the solution itself has parallels in industrial experience.

The Method

The GHS-method generates periods, deadlines and offsets, such that *if the solution is schedulable*, the implementation is guaranteed to satisfy the end-to-end timing requirements. However, the method may generate a solution which is not schedulable or it may fail to generate a solution at all – even when there is a schedulable solution. The problem then has to be transformed and the method must be applied to the transformed problem.

The method fails if:

- the constraint-set is unsatisfiable because of too tight external requirements or too high *WCET*s;
- the heuristics fail to produce a set of periods as the solution to the reduced problem, for example due to a too low maximal utilization requirement.
- all possible sets of period solutions to the reduced problem fail to result in a solution to offsets and deadlines.
- all solutions from the method to the complete problem given as periods, offset and deadlines cannot be scheduled.

None of the failures are critical or unexpected. There are two possible approaches to handle a failure:

- to change the end-to-end requirements by reapplying the preliminary design (or negotiating the control requirements) and applying the method for detailed design synthesis again *or*
- to make a transformation of the ATG [8] that conserves the satisfiability of the functional requirements and apply the method again.

In our case we had to tighten the initial (estimated) *WCET*s in order to avoid the first type of failure.

The most difficult type of failure is the last – *the method does not guarantee schedulability.* As demonstrated by this case, several additional constraints had to be added to make the solution schedulable. These con-

straints cannot easily be derived; but having a flexible tool to experiment with helped considerably.

Task Communication

Because of the asynchronous buffers the GHS-method imposes no locking between tasks accessing the same buffer. This is a major advantage of the method. If the correlation constraints are non-critical the buffers can even be reduced to a single slot as discussed in [12].

Priority Assignment and Schedulability

The GHS-method [8], assumes priority based scheduling, since the precedence constraints for intermediate tasks are guaranteed by the priority on the intermediate tasks. However, precedence can be realized in other ways and the method does not inherently assume the use of a specific method for scheduling.

An analysis for schedulability using priority based scheduling must instantiate priorities for all tasks. As the schedulability must be guaranteed for the solution to the end-to-end requirements, a method must be given to assign priorities, for example optimal priority assignment.

It can be anticipated that the schedulability analysis requires significant effort (given a task set characterized by periods, *WCET*s, deadlines and offsets) and is it not clear how to integrate the synthesis method with the analysis, except that a straightforward iterative method can be used. The problem of estimating schedulability during synthesis is not easy, to say the least. In [17] a gain factor for the response time is maximized when allocating time budgets to single tasks in a distributed real-time system, while satisfying schedulability and the end-to-end timing requirements. The gain factor is used as an estimate of individual task schedulability.

Generally it can be said that schedulability analysis for task sets containing precedence constraints and offsets is hard to do for the general case. Different exact and approximate (but conservative) methods are proposed that make different assumptions about the task set (cf. [3, 10, 21]). A systematic method to select method for schedulability analysis given a task set with certain characteristics and a scheduling method calls for further research.

The simplest method is to assume that the tasks are independent. The worst scheduling case then corresponds to simultaneous release of all tasks with higher (or equal) priority to the one for which the response time is analysed. This would, however, be too pessimistic for the general control example where the chains of computations can be long.

The Space of Solutions

Even if the method significantly reduces the space of solutions, the example can still be solved using 10^7 combinations of periods. The solution for offsets and deadlines in the example was relatively easy to find, but the worst case scenario will not be easily solved – in particular if the schedulability analysis is resource demanding. More analysis of the complexity and perhaps other heuristics should be investigated further. An alternative heuristics for the period assignment sub-problem is described in [16].

Constraint Logic Programming

To support the method we implemented the main ideas in SICStus Prolog with constraints over the rationals. The experience was encouraging – CLP offers great flexibility and the time spent on developing a working prototype was short (less than a day). Although SICStus Prolog implements constraint handling only by means of Prolog libraries the time to compute a solution was short – only 2.5 seconds on a 200MHz Pentium Pro for our example. A dedicated constraint system like CHIP (e.g. [2]) is likely to improve on this.

A problem that still lacks a satisfactory solution, however, is what *heuristics* to use when solving non-linear constraints. In our case the reduced non-linear sub-problem for the periods was reasonably small and with some user intervention (setting an upper bound on the harmonicity variables) it was possible to come up with a reasonable solution using bounded, exhaustive depth-first search. However, for larger problems it is necessary to employ more intelligent search strategies (e.g. branch-and-bound) or non-exhaustive search.

6.2. Extensions

Feedback

The original computation graph contained cycles. The cycles are typical for control systems, since state information from one cycle must be transferred to the next cycle. In the current work this was handled in an ad hoc way, by excluding some of the dependencies from the graph.

A more satisfactory approach would be to consider state as input/output data. The end-to-end timing requirements then become functions of period (instead of constant), as output from one cycle must finish before the value is used as input to a computation in the next calculation. This means that the cyclic, directed graph is transformed into an acyclic directed graph, where the output buffer of the state calculating task becomes both the input and output buffer of a computational chain – one for each use of the state variable.

Consider two tasks – τ_i with period T_i, that produces state data and τ_j with period T_j, that consumes data in the next "cycle". Since the feedback results in a cyclic graph, τ_i also consumes data produced by τ_j (perhaps after some intermediary calculations) in the same cycle. Let τ_i^0 be an instance of τ_i that consumes data produced by the instance τ_j^0. τ_i^0 then produces state data used in a later instance of τ_j. The constraint becomes a constraint on the deadline of τ_i^0, as the calculated state value must be available before it is used. The deadline constraint will correspond to the worst case for all τ_j that consumes state data produced by τ_i:

$$\forall j\{D_i \le \min(k_j T_j + \phi_j + \Phi_{ij})\}$$
$$\text{where} \quad k_j > (O_i + e_i - \phi_j - \Phi_{ij})/T_j$$

Here Φ_{ij} is the phase shift between the start of period for τ_j^0 and start of period for τ_i^0. k_j enumerates the instances of τ_j and may be treated as integer variables, but this increases the non-linearity in the equations. For $k_j = 0$ it can be assumed that the corresponding period instance for τ_j is the first instance starting after the beginnning of the period instance for τ_i^0, i.e., Φ_{ij} are all positive. To minimize, all Φ_{ij} can be assumed to be zero, as this corresponds to the worst case.

ϕ_j is the minimum delay time in a period before τ_j may start (minimum in-period phase shift). The in-period phase shift is due to the delay of τ_j caused by predecessor tasks. O_i is the offset of τ_i. Observe that it is now an output task.

In the current project there was no time to investigate this approach further since is changes the end-to-end constraints significantly so the GHS-method would not be directly applicable.

Constraint Logic Programming

The flexibility of constraint logic programming opens up new possibilities. The GHS-method strictly requires that end-to-end requirements and *WCET*s for the tasks are known – only task periods, deadlines, and offsets are allowed to be unknown. This is not always the case in an industrial setting – in the early design phases it is just as adequate to ask – "What is the maximum possible *WCET* for a certain computation?" or "What is the minimum computational delay from A to B?". For this purpose, CLP makes less assumptions about what is input and what is output to the problem. The problem formulation would be basically the same no matter what the unknowns are. However the heuristics may, of course, vary from case to case.

The GHS-method assumes a specific form for the final solution (harmonicity, no feedback), and makes no assumptions about other parameters, like the choice of scheduling method. To find the "best" design, experimentation by relaxing or varying assumptions is needed. CLP has the flexibility to support such experiments. It could, for example, be interesting to:

- investigate how feedback influences the choice of heuristics.
- investigate other heuristics for choice of "best" solution or how to handle non-linear equations, where some manual manipulation was required.
- experiment with other constraint domains and a dedicated CLP-system like CHIP. One domain that seems particularly interesting is *finite domains*; CLP(fd) contains efficient constraint solvers over finite intervals of integers and has been successfully applied to e.g. scheduling and resource allocation problems.

6.3. Related Work

The method used in this work (the period calibration method [8]) has been further extended by the original authors. In [18] the method is applied to a *distributed system*. In this work the period assignment uses fixed task allocation and similar utilization bounds as in the single processor case, but for each processor. The end-to-end deadlines then have to be allocated to each task. This is done by calculating the response times for each task and asserting the allocated deadlines to be a maximal constant factor times the response time, while the constraints on the end-to-end deadline are satisfied. Fixed priority scheduling is used to calculate the response times and the priorities are varied until a satisfactory solution is found. When the deadlines have been asserted the offsets for the tasks are calculated. The method is approximate and heuristically based.

A case study is conducted in [12], which investigates the applicability of the method and the effect of latency, periods and output jitter on the quality of the system control. A similar study of the effects of latency and sampling frequencies was made in [15]. In [19] an effort is made to optimize the quality of control by using optimal control theory and increase the processor utilization.

Our case study does not investigate the control quality, but focus the applicability of the method in existing industrial design environments. Industrial practice is that the bounds given by control design on output separation, correlation and freshness are upper bounds. Normally 2-4 samples per rise time (first and second order linear systems), i.e., $2 < T_r/h < 4$, where T_r is the rise time and h is the sampling period, is satisfactory. The analysis of the effect of end-to-end requirements on control quality and stability can be integrated into the control design [22] and reliable (but negotiable) bounds can be achieved. In this work it is assumed that the timing requirements developed

during control design are strict and that the utilization is minimized to simplify future system extensions and changes.

7. Conclusions

We have applied a method (the period calibration method [8]) for the detailed design of a task set which satisfies a set of end-to-end constraints to a typical example of avionics control software. The general conclusion is that the method does provide a feasible way to do this. The asynchronous communication is particularly appreciated.

The general approach to restrict the design architecture, for example by assuming that periods are multiples of a basic period – as opposed to assuming general periods (read: related like the primes) – to obtain a problem that is easier to solve is attractive, particularly for industrial use.

Some discrepancies where noticed in the way the problem was formulated. In particular it was noticed that the utilization is unsatisfactory as an inverse estimate of schedulability, since the method generated a solution which was easily seen to be unschedulable.

Also, the strict view on what is input and what is output of the GHS-method must be relaxed.

The project gave a positive response on the use of methods such as this for automatizing detailed design of control software, but also generated several issues that have to be investigated before actually recommending industrial production use.

Acknowledgments

We would like to thank Saab Dynamics AB for providing support and Lars-Åke Classon in particular for discussions and for supplying the specification for the case study. Thanks are also due to the anonymous reviewers and Manas Saksena for valuable comments on the paper.

References

[1] Ada 95 Reference Manual, ISO/IEC 8652:1995(E)

[2] Aggoun, A. and Beldiceanu, N. Overview of the CHIP Compiler System. In Benhamou and Colmerauer (ed), *Constraint Logic Programming: Selected Research*, 421–435, The MIT Press, 1993.

[3] Bate I., Burns A., Schedulability Analysis of Fixed priority Real-Time Systems with Offsets, in *Proc. of 9th Euromicro Workshop on Real-Time Systems*, Toledo, June 1997, IEEE Computer Society.

[4] Bonnier, S., Kalmelid, S. and Törne, A., Control System Design – An Overview of Timing Analysis Methods and Tools, restricted availability, contact the authors of the current report for authorization procedure.

[5] Burns, A., and Wellings, A. *HRT-HOOD^{TM}: A Structured Design Method for Hard Real-Time Ada Systems*. Elsevier, 1995.

[6] Burns, A., and Wellings, A. HRT-HOOD: A Structured Design Method for Hard Real-Time Systems. *Real-Time Systems*, 6:73-114, 1994.

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

[8] Gerber, R., Hong, S., and Saksena, M., Guaranteeing Real-Time Requirements With Resource-Based Calibration of Periodic Processes. *IEEE Transactions on Software Engineering*, 21(7):579 - 592, July 1995.

[9] Jaffar, J. and Maher, M. Constraint Logic Programming: A Survey. *J. Logic Programming*, 19/20:503–582, 1993.

[10] Klein, M.H., Ralya, T., Pollak, B., Obenza., R., Gonzalez Harbour, M., *A Practitioner's Handbook for Real-Time Analysis*, Kluwer Acad Publ, 1993.

[11] Kopetz, H., Zainlinger, R., Fohler, G., Kantz, H., Puschner, P., and Schutz, W. The Design of Real-Time Systems: From Specification to Implementation and Verification. *Software Engineering Journal*, 6(3): 72-82, May 1991.

[12] Kim, N., Ryu, M., Hong, S., Saksena, M., Choi, C-H. and Shin, H., Visual Assessment of a Real-Time System Design: A Case Study on a CNC-controller, In *Proc IEEE Real-Time Systems Symposium*, Dec 1996.

[13] Marriott, K. and Stuckey, P. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[14] Pospischil, G., Puschner, P., Vrchoticky, A., and Zainlinger, R. Developing Real-Time Tasks with Predictable Timing. *IEEE Software*, 35–44, September 1992.

[15] Ryu, M., Hong, S. and Saksena, M., Streamlining Real-Time Controller Design: From Performance Specifications to End-to-End Timing Constraints, In *Proc, IEEE Real-Time Technology and Applications Symposium*, Montreal, June 1997.

[16] Ryu, M. and Hong, S., A Period Calibration Algorithm for Embedded Digital Controllers. Seoul National University, SNU-EE-TR-96-1, October 1996.

[17] Saksena, M. and Hong, S., An Engineering Approach to Decomposing End-to-End Delays on a Distributed Real-Time System, In *Proc, IEEE Workshop on Parallel and Distributed Real-Time Systems*, April 1996.

[18] Saksena, M. and Hong, S., Resource Conscious design of Distributed Real-Time Systems – An End-to-End Approach, In *Proc. IEEE Int'l Conf on Engineering of Complex Computer Systems*, Oct 1996.

[19] Seto, D., Lehoczky, J., Sha, L.and Shin, K., On Task Schedulability in Real-Time Control Systems, In *Proc IEEE Real-Time Systems Symposium*, Dec 1996.

[20] Swedish Institute of Computer Science. *SICStus Prolog User's Manual*. Release 3#6. 1995.

[21] Tindell K., Adding Time-Offsets to Schedulability Analysis, Internal report, University of York, Computer Science Dept, YCS-94-221.

[22] Åström, K.J. and Wittenmark, B., *Computer Controlled Systems – Theory and Design*, Prentice Hall 1984.