UPPSALA UNIVERSITY

Department of Computer Systems

Cross-Compiling Software Circuits to CHaRy

Peter Altenbernd

Technical Report

DoCS 97/94 ASTEC 97/11

October 29, 1997

ISSN 0283-0574

Contents

1 Introduction **1**

2 Code Generation Issues **2**

2.1 Processes 3

2.2 Port and Variable Declaration 3

2.3 POP bodies 3

2.4 Edges: Signals and Precedences 4

2.5 Sensors and Actuators 5

2.6 Attachments 5

2.7 Timing Constraints 6

2.8 Network Description 6

2.9 Consistency Checks 6

2.10 Ignored Constructs 6

3 Implementation **7**

3.1 POP Library 7

3.2 Code Ordering 7

4 Conclusions **7**

A Example Compilation **8**

Abstract

This report gives a short overview of cross-compiling a text-phrase representation of **S**oftware **C**ircuits *(SCs)* to the **C**HaRy **C**ontrol **F**low **L**anguage *(CCFL)*. SCs describe processes as precedence graphs, in which each vertex represents the instance of a **P**rogrammed **OP**eration *(POP)*. CCFL is a simple PASCAL-related programming language which is used as the front-end for *CHaRy* (**C**-LAB **Ha**rd **R**eal-Time S**y**stem). The goal of CHaRy is to automatically derive feasible distributed hard real-time implementations from higher-lever descriptions. For test purposes the cross-compiler also provides C-code generation.

SCs are very well suited to model an implementation view of a hard real-time systems. By cross-compiling SCs to CCFL, this advantage can be exploited by CHaRy. In turn, it is possible to automatically derive a feasible implementation of the system specified by SCs.

1 Introduction

The purpose of this work was to develop a cross-compiler from a text-phrase representation of **S***oftware* **C***ircuits (SCs)* [4] to the **C***HaRy* **C***ontrol* **F***low* **L***anguage (CCFL)* [1].

SCs are used for modelling the software of complex real-time applications within the *Basement* system [3]. Basement is a distributed real-time architecture designed for vehicle internal communication and execution. The hardware (i.e. the processor network) is supposed to be bus interconnected.

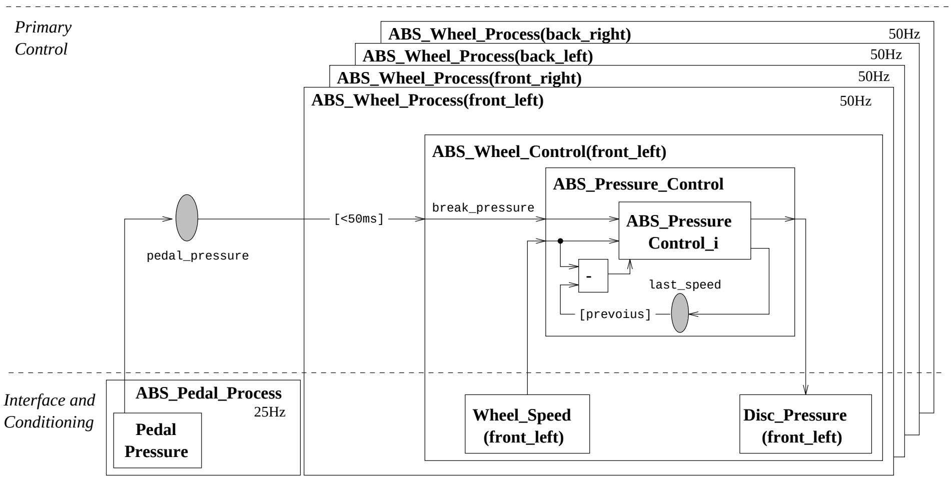

Figure 1: Software Circuits example

SCs describe periodic *processes* as precedence graphs, in which each vertex represents the instance of a **P***rogrammed* **OP***eration (POP)*. Processes represent a number of computations running with the same periodicity (sometimes referred to as a *transaction* [1]). POPs represent instances of basic computation units which can be connected in order to fulfil the required task. The directed edges in the precedence graph can be used for exchanging data values between POPs (communication). For hierarchical clustering it is possible to consider whole SCs as vertices (POPs) that can be referenced to as different instances. An example is shown in Fig. 1.

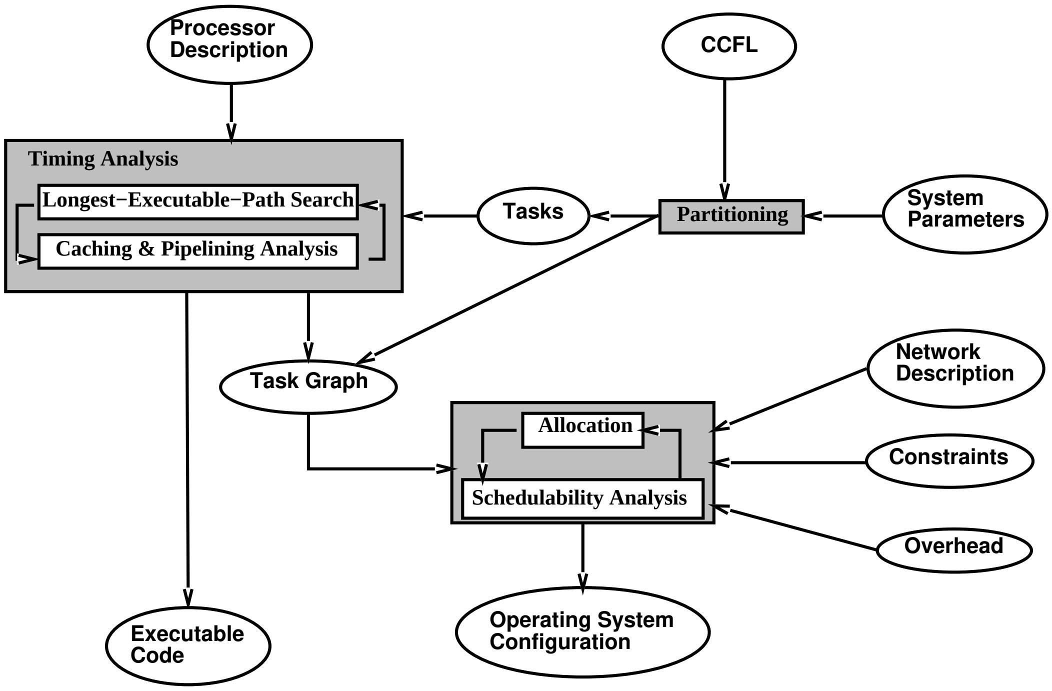

Figure 2: Overview of CHaRy

CHaRy (see Fig. 2) automatically derives distributed hard real-time implementations from higher-lever descriptions, including partitioning, timing analysis, allocation and schedulability analysis, based upon CCFL descriptions. That means, the programs, given in the PASCAL-like CCFL input format, are mapped to a distributed hardware architecture, so that all specified real-time constraints are guaranteed. CHaRy is an off-line tool that deals with periodic processes with hard deadlines, only. CHaRy can be adapted to different processor types, different operating systems (or real-time kernels), and different communication systems, and is able to deal with arbitrary networks.

The main benefits of cross-compiling SCs to CCFL is that CHaRy can exploit all advantages of this real-time dedicated high-level modelling technique. In turn, it is possible to automatically derive a feasible implementation of the system specified by SCs, including a suitable communication system and real-time kernel.

The rest of this report deals with the steps necessary to transform certain SC constructs into CCFL statements, which is mainly pointed out in Section 2. Some remarks on the implementation of the cross-compiler are given in Section 3. Conclusions are given in Section 4, in particular with regard to possible improvements to the text-phrase representation of SCs. The cross-compilation of the example shown in Fig. 2 is given in the appendix A.

2 Code Generation Issues

This section presents the SC constructs and how they are transformed to CCFL code.

2.1 Processes

Processes represent a number of computations running with the same periodicity. In the text-phrase representation of SCs, a process clusters a list of POP instances, and a list of precedences (edges).

The cross-compiler creates one CCFL program for each SC process.

2.2 Port and Variable Declaration

Ports at POPs represent connectors to be used by *signals* to exchange data. The declaration of a port in the text-phrase representation of SCs is C-like. This is cross-compiled to a PASCAL-like declaration. Further, a prefix is given to each port name in order to have unique identifiers in the generated CCFL program. The prefix is equal to the instance name of the POP. The SC data types *real*, and *integer* are converted to *REAL*, and *INTEGER*, respectively. For example, the declaration

x: **real**;

is cross-compiled to

double y_x;

provided that the corresponding instance name is y.

Additional local variables have to programmed in CCFL *templates* since variables are not available in the text-phrase representation of SCs. However, variables are seldom needed. These templates are given with regard to each POP type, and they are used for the code generation related to each POP instance. All templates are maintained in a library of POP types (see Section 2.3, and Section 3.1). When referring to a POP instance, a prefix is given to each variable name defined in the template. This is done in order to have unique identifiers in the generated CCFL program. The prefix is equal to the instance name of the POP. For example, the template

double $_x;

is cross-compiled to

double y_x;

provided that the corresponding instance name is y.

2.3 POP bodies

Templates of POP bodies have to be programmed in CCFL since they are not available in the text-phrase representation of SCs. However, POPs are usually very concise. All POP types are maintained in a library (see Section 3.1).

In order to have unique identifiers, the cross-compiler creates prefixes to all port and variable names. The prefix is equal to the instance name of the POP. For example, the template

```
IF $_speed_difference > (0.9*log(1+ $_wheel_speed/10)) THEN
BEGIN
    $_drum_pressure := 0;
END
ELSE
BEGIN
    $_drum_pressure := $_break_pressure;
END;
$_wheel_speed_old := $_wheel_speed;
```

is cross-compiled to

```
IF PC_FL_speed_difference > (0.9*log(1+ PC_FL_wheel_speed/10)) THEN
BEGIN
    PC_FL_drum_pressure := 0;
END
ELSE
BEGIN
    PC_FL_drum_pressure := PC_FL_break_pressure;
END;
PC_FL_wheel_speed_old := PC_FL_wheel_speed;
```

provided that the corresponding instance name of the POP is *PC_FL*.

2.4 Edges: Signals and Precedences

SC edges are used for exchange of data using *signals*, or for expressing precedences. Each process-internal edge is cross-compiled to a corresponding assignment statement, i.e.

```
PopId1(Port1) to PopId2(Port2)
```

is cross-compiled to

```
PopId2_Port2 := PopId1_Port1;
```

Each process-external edge is cross-compiled to corresponding send/receive statements[1], i.e.

```
PopId1(Port1) to PopId2(Port2)
```

is cross-compiled to

```
SEND(Process2, PopId1_Port1);
```

[1]not implemented yet

in the CCFL code of *Process1*, and

RECEIVE(Process1, PopId2_Port2);

in the CCFL code of *Process2*, provided that *PopId1* is part of *Process1* and *PopId2* is part of *Process2*.

For simple precedence edges containing no signals, no code is generated. The corresponding precedence is assured by the ordering within the code generation process (see Section 3.2).

Feedback edges are edges within the same process that transmit data from the previous period. For these edges the same code is generated as for normal edges. However, when ordering the code, feedback edges are ignored (otherwise no ordering would be possible due to cyclic dependences). This imposes a situation in which a variable/port is used before it is set. Hence, the corresponding CCFL variable is defined to be *static* (according to C-semantics). Those CCFL variables are initialised with value *zero* since there is no way in the text-phrase representation of SCs to define different initialisation values.

2.5 Sensors and Actuators

POPs with no predecessor in the precedence graph are considered to be *sensors*. For each port P in a sensor instance S the following CCFL code is generated:

READ S_P;

POPs with no successor in the precedence graph are considered to be *actuators*. For each port P in an actuator instance A the following CCFL code is generated:

WRITE A_P;

2.6 Attachments

Certain POP instances can be attached to certain processors due to input/output access to special devices. The code of an attached POP instances is nested into a corresponding CCFL statement. For example, the attachment

POP_type : POP **ON** processor

is cross-compiled to

ATTACHMENT TO processor
BEGIN
(code of POP *) ...*
END;

2.7 Timing Constraints

Period lengths and deadlines in the text-phrase representation of SCs can be given in different time units. Further, these specifications may also be given by frequencies using different units. Also these units are converted to micro seconds for input to CHaRy. These specifications are not part of CCFL but of the definition of timing *constraints* (see Fig. 2).

2.8 Network Description

The text-phrase representation of SCs assumes a network of identical processors inter-connected by a bus. The network is described simply by listing the processor names. These names are used to create the corresponding *network description* (see Fig. 2) needed for CHaRy (which also allows the use of networks different from bus-based systems).

2.9 Consistency Checks

The cross-compiler also includes a number of simple consistency checks which are listed below:

- Check on multiple declaration of POP types
- Check whether all used POP types are known and implemented
- Check on multiple declaration of POP instances
- Check on multiple declaration of port identifiers
- Check on multiple declaration of process identifiers
- Check on multiple declaration of processor identifiers
- Check whether all POP instance identifiers used for edges are known
- Check whether all port names used for edges are known in the corresponding context
- Check whether all data exchanging edges contain the required port names
- Check on cyclic dependences (due to feedback edges)
- Check whether all processor identifiers used for attachments are known

2.10 Ignored Constructs

All constructs related of timing analysis, scheduling, and allocation are ignored during cross-compilation, since they are automatically handled by CHaRy. These are: the *start* time of a POP instance, the *AGE* of an edge, the *NUMBER_OF_NODES*, the *FRAME_TIME*, the *CAN_RECOVER_TIME*, the *CAN_INIT_TIME*, the *CAN_ABORT_TIME*, the number of *TICKS_PER_SEC*, the *NUMBER_OF_POPS*, the *SLOT_TIME*, the *FRAMES_PER_SLOT*, and all *EXECUTION_TIMES*.

3 Implementation

The cross-compiler was implemented using the compiler tool kit *Eli*[2] [2]. The Eli system creates executable text processing programs from specifications. A user provides Eli with specifications that describe a particular text processing problem, and Eli creates a program to solve that problem. All auxiliary programs for use within the Eli specifications were written in C++. For the representation of the precedence graph the C++ library *LEDA*[3] *(Library of Efficient Data types and Algorithms)* [5] was used.

All sources implementing the cross-compiler can be found in
`http://www.docs.uu.se/~petalt/codegen`.

3.1 POP Library

The templates of POP types as CCFL code have to be programmed in a file called `pops.ptg`. This files contains specifications according to Eli's *Pattern-based Text Generator* (PTG). More detailed information on how to include new POP types into this file can be found in the comments at the beginning of file `pops.ptg`.

3.2 Code Ordering

To achieve the correct ordering of the CCFL code, the code generation visits all POP instances in the *topological order* according to the precedences defined by the edges. This is programmed in file `codegen.c`.

4 Conclusions

This report has given a short overview of issues related to cross-compiling a text-phrase representation of SCs to CCFL. SCs are very well suited to model an implementation view of a hard real-time systems. By coupling SCs to CHaRy it is possible to automatically produce controller implementations.

However, the text-phrase representation of SC should be revised with regard to the following issues:

1. Scope Rules. Due to the way edges going from one process to another process is described, all POP instance identifiers have to be defined within the same, global scope, although they are clustered into processes. In order to exploit the use of local scopes for name-overloading the definition of SC edges should include process names, i.e. an explicit definition of *export* or *import* edges.

2. Initial Values for Feedbacks. For feedback edges there is a need of having an initial value for the first iteration for the corresponding variable. Per default this is zero (see

[2]More information about Eli can be found in
`http://www.uni-paderborn.de/fachbereich/AG/agkastens/elionline/index.html`

[3]More information about LEDA can be found in
`ftp://ftp.mpi-sb.mpg.de/pub/LEDA/index.html`

Section 2.4). However, the system behaviour might be totally different, if a different initial value is used. Hence, a concept how to include this into the SC description is needed.

3. Clear Encapsulation of SW. The text-phrase representation of SC includes many non-SW related issues, like the network description, worst-case execution times, etc.. All these things should be separated from the actual software in order to have a clear distinction between the software, and system parameters.

4. Templates for POP bodies. A more sophisticated method than using PTG for maintaining a library of POP types is needed.

References

[1] Peter Altenbernd. *CHaRy: The C-LAB Hard Real-Time System to Support Mechatronical Design.* In *Proceedings of the International Conference and Workshop on Engineering of Computer Based Systems ECBS'97*, pages 271–278, 1997.

[2] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. *Eli: A Complete, Flexible Compiler Construction System Search. Communications of the ACM*, 35(2):121–131, Februar 1992.

[3] H. Hansson, H. Lawson, M. Strömberg, and S. Larsson. BASEMENT a distributed real-time architecture for vehicle applications. *Real-Time Systems*, 11(3):223–244, November 1996.

[4] E. Larsson. *BASEMENT Real-Time System: The Scheduling Tool.* Technical Report ProVIA-94204, Department of Computer Systems, Uppsala University, 1995.

[5] K. Mehlhorn and S. Näher. *LEDA, a Platform for Combinatorial and Geometric Computing. Communications of the ACM*, 38(1):96–102, 1995.

[6] Friedhelm Stappert. *Predicting Pipelining and Caching Behaviour of Hard Real-Time Programs.* In *Proceedings of the 9th Euromicro Workshop on Real-time Systems*, 1997.

A Example Compilation

In the following the compilation of the example in Fig.1 is given. The presentation is focused on the software of one process. Further, all ignored constructs are not considered.

```
// ====================================================================
// POPs and Ports:
// Here all POPs available to the applications are defined.
// The description includes: POP name, port names with direction and type.
// --------------------------------------------------------------------
// Application POPs
```

```
POP Pressure_Control_i
  INPUT
    break_pressure   : real
    wheel_speed      : real
    speed_difference : real
  OUTPUT
    drum_pressure    : real
    wheel_speed_old  : real
END_POP

// ------------------------------
// Sensor/Actuator POPs


POP Break_Pedal_Pressure
  OUTPUT
    pressure : real
END_POP

POP Wheel_Speed
  OUTPUT
    speed : real
END_POP

POP Drum_Pressure
  INPUT
    pressure : real
END_POP


// ------------------------------
// System defined POPs

POP minus
  INPUT
    term1 : real
    term2 : real
  OUTPUT
    difference : real
END_POP



PROCESS ABS_Process(PERIOD 25ms DEADLINE 25ms)
  POPS
    ABS_Break_Pedal_Pressure: Break_Pedal_Pressure     ON driver

    Wheel_Speed_FL          : Wheel_Speed              ON front_left
    Drum_Pressure_FL        : Drum_Pressure            ON front_left
    PC_FL                   : Pressure_Control_i
    minus_FL                : minus

    Wheel_Speed_FR          : Wheel_Speed              ON front_right
    Drum_Pressure_FR        : Drum_Pressure            ON front_right
    PC_FR                   : Pressure_Control_i
    minus_FR                : minus


    Wheel_Speed_BL          : Wheel_Speed              ON back_left
    Drum_Pressure_BL        : Drum_Pressure            ON back_left
```

```
    PC_BL                   : Pressure_Control_i
    minus_BL                : minus

    Wheel_Speed_BR          : Wheel_Speed             ON back_right
    Drum_Pressure_BR        : Drum_Pressure           ON back_right
    PC_BR                   : Pressure_Control_i
    minus_BR                : minus
  END_POPS

  PRECEDENCE
    ABS_Break_Pedal_Pressure(pressure) to PC_FL(break_pressure)
    ABS_Break_Pedal_Pressure(pressure) to PC_FR(break_pressure)
    ABS_Break_Pedal_Pressure(pressure) to PC_BL(break_pressure)
    ABS_Break_Pedal_Pressure(pressure) to PC_BR(break_pressure)

    Wheel_Speed_FL(speed)  to minus_FL(term1)
    Wheel_Speed_FL(speed)  to PC_FL(wheel_speed)
    minus_FL(difference)   to PC_FL(speed_difference)
    PC_FL(drum_pressure)   to Drum_Pressure_FL(pressure)
    PC_FL(wheel_speed_old) to minus_FL(term2)    [FEEDBACK]

    Wheel_Speed_FR(speed)  to minus_FR(term1)
    Wheel_Speed_FR(speed)  to PC_FR(wheel_speed)
    minus_FR(difference)   to PC_FR(speed_difference)
    PC_FR(drum_pressure)   to Drum_Pressure_FR(pressure)
    PC_FR(wheel_speed_old) to minus_FR(term2)    [FEEDBACK]

    Wheel_Speed_BL(speed)  to minus_BL(term1)
    Wheel_Speed_BL(speed)  to PC_BL(wheel_speed)
    minus_BL(difference)   to PC_BL(speed_difference)
    PC_BL(drum_pressure)   to Drum_Pressure_BL(pressure)
    PC_BL(wheel_speed_old) to minus_BL(term2)    [FEEDBACK]

    Wheel_Speed_BR(speed)  to minus_BR(term1)
    Wheel_Speed_BR(speed)  to PC_BR(wheel_speed)
    minus_BR(difference)   to PC_BR(speed_difference)
    PC_BR(drum_pressure)   to Drum_Pressure_BR(pressure)
    PC_BR(wheel_speed_old) to minus_BR(term2)     [FEEDBACK]
  END_PRECEDENCE
END_PROCESS
```

The resulting CCFL program is:

```
COMPLEX eval
VAR
(* Declarations: *)
(* ABS_Break_Pedal_Pressure *)
    ABS_Break_Pedal_Pressure_pressure : REAL := 0;

(* Wheel_Speed_FL *)
    Wheel_Speed_FL_speed : REAL := 0;

(* Wheel_Speed_FR *)
    Wheel_Speed_FR_speed : REAL := 0;

(* Wheel_Speed_BL *)
    Wheel_Speed_BL_speed : REAL := 0;
```

```
(* Wheel_Speed_BR *)
    Wheel_Speed_BR_speed : REAL := 0;

(* minus_FL *)
    minus_FL_term1 : REAL := 0;
    minus_FL_term2 : REAL := 0;
    minus_FL_difference : REAL := 0;

(* minus_FR *)
    minus_FR_term1 : REAL := 0;
    minus_FR_term2 : REAL := 0;
    minus_FR_difference : REAL := 0;

(* minus_BL *)
    minus_BL_term1 : REAL := 0;
    minus_BL_term2 : REAL := 0;
    minus_BL_difference : REAL := 0;

(* minus_BR *)
    minus_BR_term1 : REAL := 0;
    minus_BR_term2 : REAL := 0;
    minus_BR_difference : REAL := 0;

(* PC_FL *)
    PC_FL_break_pressure : REAL := 0;
    PC_FL_wheel_speed : REAL := 0;
    PC_FL_speed_difference : REAL := 0;
    PC_FL_drum_pressure : REAL := 0;
    PC_FL_wheel_speed_old : REAL := 0;

(* PC_FR *)
    PC_FR_break_pressure : REAL := 0;
    PC_FR_wheel_speed : REAL := 0;
    PC_FR_speed_difference : REAL := 0;
    PC_FR_drum_pressure : REAL := 0;
    PC_FR_wheel_speed_old : REAL := 0;

(* PC_BL *)
    PC_BL_break_pressure : REAL := 0;
    PC_BL_wheel_speed : REAL := 0;
    PC_BL_speed_difference : REAL := 0;
    PC_BL_drum_pressure : REAL := 0;
    PC_BL_wheel_speed_old : REAL := 0;

(* PC_BR *)
    PC_BR_break_pressure : REAL := 0;
    PC_BR_wheel_speed : REAL := 0;
    PC_BR_speed_difference : REAL := 0;
    PC_BR_drum_pressure : REAL := 0;
    PC_BR_wheel_speed_old : REAL := 0;

(* Drum_Pressure_FL *)
    Drum_Pressure_FL_pressure : REAL := 0;

(* Drum_Pressure_FR *)
    Drum_Pressure_FR_pressure : REAL := 0;

(* Drum_Pressure_BL *)
    Drum_Pressure_BL_pressure : REAL := 0;
```

```
(* Drum_Pressure_BR *)
    Drum_Pressure_BR_pressure : REAL := 0;

(* eval code: *)
BEGIN
  (* ABS_Break_Pedal_Pressure *)
ATTACHMENT TO driver
BEGIN
  READ ABS_Break_Pedal_Pressure_pressure; (* x[0] *)
END;

  PC_FL_break_pressure := ABS_Break_Pedal_Pressure_pressure;
  PC_FR_break_pressure := ABS_Break_Pedal_Pressure_pressure;
  PC_BL_break_pressure := ABS_Break_Pedal_Pressure_pressure;
  PC_BR_break_pressure := ABS_Break_Pedal_Pressure_pressure;

  (* Wheel_Speed_FL *)
ATTACHMENT TO front_left
BEGIN
  READ Wheel_Speed_FL_speed; (* x[1] *)
END;

  minus_FL_term1 := Wheel_Speed_FL_speed;
  PC_FL_wheel_speed := Wheel_Speed_FL_speed;

  (* Wheel_Speed_FR *)
ATTACHMENT TO front_right
BEGIN
  READ Wheel_Speed_FR_speed; (* x[2] *)
END;

  minus_FR_term1 := Wheel_Speed_FR_speed;
  PC_FR_wheel_speed := Wheel_Speed_FR_speed;

  (* Wheel_Speed_BL *)
ATTACHMENT TO back_left
BEGIN
  READ Wheel_Speed_BL_speed; (* x[3] *)
END;

  minus_BL_term1 := Wheel_Speed_BL_speed;
  PC_BL_wheel_speed := Wheel_Speed_BL_speed;

  (* Wheel_Speed_BR *)
ATTACHMENT TO back_right
BEGIN
  READ Wheel_Speed_BR_speed; (* x[4] *)
END;

  minus_BR_term1 := Wheel_Speed_BR_speed;
  PC_BR_wheel_speed := Wheel_Speed_BR_speed;

  (* minus_FL *)
  minus_FL_difference := minus_FL_term1 - minus_FL_term2;

  PC_FL_speed_difference := minus_FL_difference;

  (* minus_FR *)
  minus_FR_difference := minus_FR_term1 - minus_FR_term2;
```

```
PC_FR_speed_difference := minus_FR_difference;

(* minus_BL *)
minus_BL_difference := minus_BL_term1 - minus_BL_term2;

PC_BL_speed_difference := minus_BL_difference;

(* minus_BR *)
minus_BR_difference := minus_BR_term1 - minus_BR_term2;

PC_BR_speed_difference := minus_BR_difference;

(* PC_FL *)
IF PC_FL_speed_difference > (0.9*log(1+ PC_FL_wheel_speed/10)) THEN
BEGIN
   PC_FL_drum_pressure := 0;
END
ELSE
BEGIN
   PC_FL_drum_pressure := PC_FL_break_pressure;
END;
PC_FL_wheel_speed_old := PC_FL_wheel_speed;

Drum_Pressure_FL_pressure := PC_FL_drum_pressure;
minus_FL_term2 := PC_FL_wheel_speed_old;

(* PC_FR *)
IF PC_FR_speed_difference > (0.9*log(1+ PC_FR_wheel_speed/10)) THEN
BEGIN
   PC_FR_drum_pressure := 0;
END
ELSE
BEGIN
   PC_FR_drum_pressure := PC_FR_break_pressure;
END;
PC_FR_wheel_speed_old := PC_FR_wheel_speed;

Drum_Pressure_FR_pressure := PC_FR_drum_pressure;
minus_FR_term2 := PC_FR_wheel_speed_old;

(* PC_BL *)
IF PC_BL_speed_difference > (0.9*log(1+ PC_BL_wheel_speed/10)) THEN
BEGIN
   PC_BL_drum_pressure := 0;
END
ELSE
BEGIN
   PC_BL_drum_pressure := PC_BL_break_pressure;
END;
PC_BL_wheel_speed_old := PC_BL_wheel_speed;

Drum_Pressure_BL_pressure := PC_BL_drum_pressure;
minus_BL_term2 := PC_BL_wheel_speed_old;

(* PC_BR *)
IF PC_BR_speed_difference > (0.9*log(1+ PC_BR_wheel_speed/10)) THEN
BEGIN
   PC_BR_drum_pressure := 0;
END
ELSE
```

```
  BEGIN
     PC_BR_drum_pressure := PC_BR_break_pressure;
  END;
  PC_BR_wheel_speed_old := PC_BR_wheel_speed;

  Drum_Pressure_BR_pressure := PC_BR_drum_pressure;
  minus_BR_term2 := PC_BR_wheel_speed_old;

  (* Drum_Pressure_FL *)
ATTACHMENT TO front_left
BEGIN
  WRITE Drum_Pressure_FL_pressure; (* dx[0] *)
END;


  (* Drum_Pressure_FR *)
ATTACHMENT TO front_right
BEGIN
  WRITE Drum_Pressure_FR_pressure; (* dx[1] *)
END;


  (* Drum_Pressure_BL *)
ATTACHMENT TO back_left
BEGIN
  WRITE Drum_Pressure_BL_pressure; (* dx[2] *)
END;

  (* Drum_Pressure_BR *)
ATTACHMENT TO back_right
BEGIN
  WRITE Drum_Pressure_BR_pressure; (* dx[3] *)
END;

END.
```

By running CHaRy with the generated CCFL program as input, the system is partitioned (see Fig. 2) into five different OS tasks whose properties are given in Tab. 1.

The estimation of the worst-case execution time was performed with regard to PowerPC CPUs [6]. Since all tasks are already attached, no allocation has to be found by CHaRy. The schedulability analysis reported the task assignments to be feasible (due to the very small processing times provided by the PowerPC).

The created tasks *task45*, and *task26* are as follows:

```
TASK task26
VAR
  PC_FL_speed_difference : REAL := 0;
  minus_FL_difference : REAL := 0;
  minus_FL_term1 : REAL := 0;
  minus_FL_term2 : REAL := 0;
  PC_FL_wheel_speed_old : REAL := 0;
  Wheel_Speed_FL_speed : REAL := 0;
  PC_FL_wheel_speed : REAL := 0;
  Drum_Pressure_FL_pressure : REAL := 0;
  PC_FL_drum_pressure : REAL := 0;
  PC_FL_break_pressure : REAL := 0;
```

name	POPs involved	worst-case exec. time	attached to
task45	ABS_Break_Pedal_Pressure	180μs	driver
task26	Wheel_Speed_FL Drum_Pressure_FL PC_FL minus_FL	1128μs	front_left
task14	Wheel_Speed_FR Drum_Pressure_FR PC_FR minus_FR	1128μs	front_left
task8	Wheel_Speed_BL Drum_Pressure_BL PC_BL minus_BL	1128μs	front_left
task6	Wheel_Speed_BR Drum_Pressure_BR PC_BR minus_BR	1128μs	front_left

Table 1: The task set created by CHaRy

```
BEGIN
  RECEIVE task45, PC_FL_break_pressure;

  READ Wheel_Speed_FL_speed;

  PC_FL_wheel_speed := Wheel_Speed_FL_speed;
  PC_FL_wheel_speed_old := PC_FL_wheel_speed;

  minus_FL_term1 := Wheel_Speed_FL_speed;
  minus_FL_term2 := PC_FL_wheel_speed_old;
  minus_FL_difference := minus_FL_term1 - minus_FL_term2;

  PC_FL_speed_difference := minus_FL_difference;

  IF PC_FL_speed_difference > (0.9 * log(1 + PC_FL_wheel_speed / 10)) THEN
  BEGIN
    PC_FL_drum_pressure := 0;
  END (* if *)
  ELSE
  BEGIN
    PC_FL_drum_pressure := PC_FL_break_pressure;
  END; (* else *)

  Drum_Pressure_FL_pressure := PC_FL_drum_pressure;

  WRITE Drum_Pressure_FL_pressure;
END.
```

```
TASK task45
VAR
  ABS_Break_Pedal_Pressure_pressure : REAL := 0;
  PC_BR_break_pressure : REAL := 0;
  PC_BL_break_pressure : REAL := 0;
  PC_FR_break_pressure : REAL := 0;
  PC_FL_break_pressure : REAL := 0;
BEGIN
  READ ABS_Break_Pedal_Pressure_pressure;

  PC_BR_break_pressure := ABS_Break_Pedal_Pressure_pressure;
  PC_BL_break_pressure := ABS_Break_Pedal_Pressure_pressure;
  PC_FR_break_pressure := ABS_Break_Pedal_Pressure_pressure;
  PC_FL_break_pressure := ABS_Break_Pedal_Pressure_pressure;

  SEND task6,  PC_BR_break_pressure;
  SEND task26, PC_FL_break_pressure;
  SEND task8,  PC_BL_break_pressure;
  SEND task14, PC_FR_break_pressure;
END.
```