MODELLING AND ANALYSIS OF BROADCASTING EMBEDDED CONTROL SYSTEMS

D.Kendall W.D.Henderson A.P.Robson

Department of Computing, University of Northumbria at Newcastle, Ellison Place, Newcastle upon Tyne, NE1 8ST; Tel: +44 191 227 3512; Email:{david.kendall, william.henderson, adrian.robson}@unn.ac.uk

1 Introduction

This paper introduces a framework for the development, modelling and analysis of distributed, real-time control systems which communicate using the deterministic broadcast communication protocol, CAN. We adopt a hierarchical approach in which system designs are expressed in the high-level, Ada-like, language, *CANDLE*, which is given a timed transition semantics by translation to a base language, *bCANDLE* (pronounced 'basic candle') which is a simple but expressive process language with a value-passing, broadcast communication primitive, message priorities and an explicit time construct. The formal semantics of *bCANDLE* can be found in [6].

Broadcast communication is used frequently in the implementation of embedded systems, but has received comparatively little attention from the formal methods community in contrast to point-to-point synchronous communication. Timed transition systems have proved to be very successful models for the analysis of real-time systems [4] and they arise naturally from a variety of formalisms for system description. We argue in [6] that there is a need for an approach to the description of broadcasting systems which adopts a broadcast mechanism as its communication primitive, with the intention of facilitating the construction of a timed transition system model which can be simulated and analysed.

We wish to promote the production of formal system models which arise almost as a by-product of a 'natural' development process. A model is intended to be a conservative approximation of its associated implementation, i.e. the behaviours of the implementation should be a subset of the behaviours of the model. Given such a conservative approximation, by restricting attention to requirements which are expressed as properties of *all* behaviours of the model, it is sufficient to establish that the model satisfies a requirement in order to conclude that the implementation also satisfies the requirement. This approach is similar in some respects to the timing analysis of Ada programs undertaken by Corbett [2]. However, the work described there is restricted to single processor systems, whereas we are concerned primarily with distributed systems.

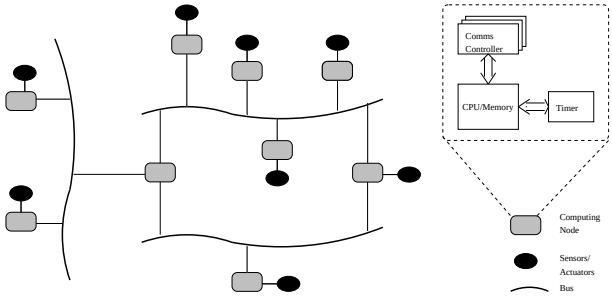

Figure 1: Control system model

2 Informal control system model

Figure 1 shows a typical organisation for the class of control systems to be studied. A number of tasks may be allocated to each computing node and will share the processing unit using some fixed scheduling policy. In order to simplify the model and to facilitate system reorganization, we assume that all tasks communicate using (logically) a single mechanism, whether they share a computing node or not. So even tasks which share a processor, communicate by broadcasting messages and do not have unconstrained access to shared memory. In addition, each computing node may have access to a number of sensors and actuators which form part of the interface to the controlled system. It is required that sensors and actuators are not shared but that each is accessed by a single task.

We have targetted our development approach at a specific communication network, namely Controller Area Network (CAN). CAN uses a simple, deterministic, broadcast communication protocol which makes it not only attractive to developers but also amenable to formal modelling and analysis. It is gaining increasing importance and attention in the implementation of distributed real-time systems [5].

3 Distributed Robot Controller

We illustrate the construction of a timed transition model for a CAN-based system using the example of a distributed robot controller which has been discussed in [2] and elsewhere. Although only a simple system, it allows the demonstration of most of the

features of *CANDLE* including its languages and approach to development and verification.

The system requires commands to be communicated to a robot from time to time. Each command is computed based upon the readings delivered by two sensors. We assume an implementation which uses three distributed tasks executing in parallel and communicating via a CAN. The tasks interact with the robot using a pair of sensors and a single actuator. There is a task responsible for reading each of the sensors and a further task to integrate the readings and send a command to the robot. The interaction with sensors and actuators is modelled and implemented by simple sequential operations (e.g. `Sensor1.ReadSensor`).

The main requirement of the system is that the command which is sent to the robot must be based upon readings received from each of the sensors with a maximum separation between the times of the readings. It is the job of the Integrator task to receive the sensor readings, compute a command and send a signal to the robot. It should be able to receive readings from the two sensors in either order. In order to satisfy the maximum separation requirement, following the receipt of the first sensor reading, the integrator task waits for only a bounded length of time for the second reading to arrive. Figure 2 illustrates the use of *CANDLE* to describe the main features of the system and the implementation of one of the sensor tasks.

The sensor tasks are activated periodically. At regular intervals, they take a sensor reading and broadcast it until an acknowledgement is received. This ensures that a fresh sensor reading is available to the integrator task. The integrator task repeatedly waits to receive a reading from either sensor and then waits for a limited period for the other sensor reading. If this reading arrives in time, the task uses both readings to compute a command which it then signals to the robot; otherwise the task tries again to receive both readings within the maximum separation distance.

Data clauses in *CANDLE* (such as `with data Sensor1`) establish a link to data specifications and implementations which are constructed using a suitable external language. Data abstraction and the extraction of state transformers from specifications is performed 'by hand'; we are investigating the use of PVS to support this process. Bounds upon the performance of data operations are obtained with the help of a C code timing tool, in conjunction with a simple multi-tasking scheduling analysis as described in [1].

A system description in *CANDLE* is used as the primary source *both* for the generation of system code *and* for the generation of a model for simulation and verification, in keeping with the spirit of WYVIWYE[1].

```
system DRC is
  Sensor1 | Sensor2 | Integrator
where
  visible
    Sensor1.ReadSensor, Sensor2.ReadSensor,
    Integrator.Signal

  network is
    channel is <ack1, ack2, sensor1, sensor2>
  end_network

  task Sensor1 with data Sensor1
  using
    constant SENSOR1_PERIOD, SENSOR1_EXPIRE
    var val
    op ReadSensor
  is
    every SENSOR1_PERIOD do
      loop DELIVER do
        ReadSensor; snd(sensor1,val);
        select
          when rcv(ack1) do exit DELIVER
        or
          when elapse SENSOR1_EXPIRE do skip
        end_select
      end_loop DELIVER
    end_every
  end_task

/* task Sensor2 ... similar to Sensor1 */

/*  task Integrator ... */

end_system
```

Figure 2: Outline of *CANDLE* system file for distributed robot controller

4 Constructing a timed transition model

A *CANDLE* system description must be translated into *bCANDLE* before its behaviour can be simulated or verified. We use the distributed robot controller example to introduce informally the translation and to illustrate salient points.

A *bCANDLE* model represents the state and behaviour of tasks and network channels. The behaviour of the model follows a two phase pattern, as discussed in [4], in which instantaneous action transitions are interspersed with time transitions in which time advances in all components. The model is constructed from a number of development files which are described in table 1.

Figure 3 gives the *bCANDLE* model for the dis-

[1] What You Verify Is What You Execute

```
Sensor1 | Sensor2 | Integrator

where

Sensor1 =
  [pre_timer];
  (Deliver [> exit_DELIVER -> [POST_EXIT_DELIVER];idle)
  [>
  [approx_SENSOR1_PERIOD];[post_timer];[jump];Sensor1

Deliver =
  [ReadSensor];
  [pre_snd]; k!sensor1._; [post_snd];
  [pre_select1];
  (k?ack1._ ; [post_rcv]; [PRE_EXIT_DELIVER] ; idle
  +
  [approx_SENSOR1_EXPIRE] ; [post_timer]
  );
  [jump] ; Deliver

/* Sensor2 = ... similar to Sensor1 */

Integrator =
  Gather [> exit_GATHER -> [POST_EXIT_GATHER];
  [Compute]; [Signal]; [jump]; Integrator

Gather =
  [pre_select2];
  (k?sensor1._;
     [post_rcv]; [pre_select3];
     (k?sensor2._;
        [post_rcv]; [pre_snd]; k!ack1._; [post_snd];
        [pre_snd]; k!ack2._; [post_snd];
        [PRE_EXIT_GATHER]; idle
     +
      [approx_PROXIMITY_MAX];
        [post_timer]
     )
  +
   k?sensor2._;
     [post_rcv]; [pre_select4];
     (k?sensor1._;
        [post_rcv]; [pre_snd]; k!ack1._; [post_snd];
        [pre_snd]; k!ack2._; [post_snd];
        [PRE_EXIT_GATHER]; idle
     +
      [approx_PROXIMITY_MAX];
        [post_timer]
     )
  );
  [jump]; Gather

network
/*             pi dlb  dub dlB duB         */
  k = (ack1:    1, 37,  47, 10, 12;
       ack2:    2, 37,  47, 10, 12;
       sensor1: 3, 43,  53, 10, 12;
       sensor2: 4, 43,  53, 10, 12)

data
  _ = @
  __exit_DELIVER = false
  __exit_GATHER = false
```

Figure 3: *bCANDLE* model of Distributed Robot Controller

`.ds`	Specification files for the data state and sequential operations of of each system task. Model-based specification languages such as B, Z or VDM can be used. Specifications are used to develop sequential code following a standard methodology and are also used to develop abstract data specifications for system verification.
`.can`	*CANDLE* system file: contains a description of the dynamic behaviour of tasks including communication and synchronisation. Declares broadcast channels, including message identifiers and their priorities.
`.sa`	System architecture file: maps tasks to processors, communication channels to CAN buses, *CANDLE* data to specifications and implementations, etc.
`.cd`	Component description files: describes the properties of system components, e.g. processors, CAN buses and clocks in order to allow the prediction of timing properties.
`.c, .o`	C source and object files developed from data specification using a standard development methodology.
`.bc`	*bCANDLE* file: low-level system model with formal timed transition semantics. Generated automatically from input files.
`.tr`	Trace file which is either output by the simulator as a history of a simulation run or which can be used as input to the simulator to guide a simulation session.
`.tg`	Timed graph file: suitable for input to external model checkers such as KRONOS and UPPAAL.
`.ts`	Temporal specification file: a specification of temporal system properties either using a logic (such as TCTL [4]) acceptable to model checker or given by a description of a specification automaton.

Table 1: *CANDLE* development files

tributed robot controller. It comprises 3 sections, defining the behaviour of system tasks, network parameters and initial data state. Task behaviour is defined in a number of possibly recursive equations using a simple process language which is summarised in table 2.

The construction of the model is based mainly upon the system description (`.can` file) but also relies upon information derived from the other system files. The code for each sequential operation is analysed to determine the bounds (i.e. the estimated best case and worst case execution times) on its ex-

`k!i.x`	Enqueue a message with identifier i and value given by x for transmission on channel k. Non-blocking.
`k?i.x`	Await a message with identifier i on channel k, store the transmitted value in x. Blocking.
`[Op:t1,t2]`	Transform the data state according to operation Op within the bounds given by t1 and t2.
`p -> T`	Evaluate the predicate p in the current data state, if true then behave as T, otherwise idle.
`T1 ; T2`	Sequential composition: behave as T1 then T2.
`T1 + T2`	Choice: choose whichever branch has a possible action transition. Network and time transitions do not resolve choice.
`T1 [> T2`	Interrupt: behave as T1 until T2 can make an action transition, then behave as `T2`. If `T1` terminates then `T1 [> T2` terminates.
`T1 | T2`	Parallel composition: asynchronous interleaving of action transitions. Synchronous time steps.

Table 2: *bCANDLE* language summary

ecution. It is necessary to know the architecture of the node on which the task is to be executed in order to perform the analysis. In the case of a multitasking node, the execution time bounds must be converted into response time bounds. This analysis is possible for a simple time-slicing scheduler [1]. In figure 3, every use of `[...]` represents a computation whose time bounds, denoted by the enclosed symbolic name, are the bounds on the response time for the corresponding operation. So, for example, `[Compute]` represents a computation whose bounds are the calculated response time bounds for the operation `Compute`.

Each communication, `snd(id,x)` or `rcv(id,x)`, requires some computation time both before and after it (to allow for delays caused by configuring a communication controller or handling an interrupt, for example). Let `[pre_snd], [post_snd]` represent the bounds on the before and after delays for a `snd`, and `[pre_rcv], [post_rcv]` the corresponding delays for `rcv`. The calculation of these bounds requires knowledge of the kernel implementation and the hardware platform.

The modelling of timer services (whose use is implied by the periodic behaviour of `ReadSensor` requires similar information regarding their low-level implementation. We use `[pre_timer], [approx Time]` and `[post_timer]` to denote the bounds on the set up time, resolution and recovery time, respectively, given by a request for a delay of `Time` time units.

`[jump]` denotes the bounds required for the execution of a jump instruction.

5 Conclusions

This paper describes work which has been undertaken as part of an ongoing project to provide a tool-supported engineering environment for the development of distributed embedded control systems.We aim to build systems in such a way that it is possible to extract abstract models which preserve important features of the quantitative properties of their implementations. These models are amenable to a variety of well-developed, tool-supported analysis techniques [4, 3, 7]. We believe that CAN will be an important component in many small/medium scale embedded control systems, because of its properties of robustness and predictability. The approach taken by *CANDLE* is intended to improve the quality of CAN control systems by enabling a straight forward construction of tractable system models. The major obstacle, as always, remains the management of the state explosion problem. We are currently seeking to apply symbolic and partial order techniques to the simplification of our system models.

References

[1] S Bradley, W Henderson, D Kendall, and A Robson. A formally based hard real-time kernel. *Microprocessors and Microsystems*, 18(9):513–521, November 1994.

[2] J.C. Corbett. Timing analysis of Ada tasking programs. *IEEE Transactions on Software Engineering*, 22(7):461–483, July 1996.

[3] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Proc. Int. Conf. on Formal Description Techniques VII (FORTE'94)*, pages 227–242, 1994.

[4] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[5] ISO/DIS 11898: Road Vehicles – interchange of digital information – Controller Area Network (CAN) for high speed communication, 1992.

[6] D. Kendall, S. Bradley, W. Henderson, and A. Robson. *bCANDLE*: Formal modelling and analysis of CAN control systems. In *Proceedings of 4th IEEE Real Time Technology and Applications Symposium (RTAS'98)*. IEEE Computer Society Press, June 1998. (to appear).

[7] K. Larsen, P. Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Springer International Journal on Software Tools for Technology Transfer*, October 1997.