CANDLE: A HIGH LEVEL LANGUAGE AND DEVELOPMENT ENVIRONMENT FOR HIGH INTEGRITY CAN CONTROL SYSTEMS

D.Kendall S.P.Bradley[1] 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] Department of Computing Science, University of Durham

Abstract

Embedded control systems appear in many of the manufactured products upon which our society increasingly depends. System developers need better development methods in order to be more confident that the systems which they deliver will behave properly. The need is particularly pressing in the case of distributed, hard real-time control systems for which testing is notoriously difficult. In recent years, much research has been conducted into formal techniques for analysing the quantitative temporal properties of system models. Such work offers the promise of complementing testing in the validation of systems by approaches which include simulation, symbolic monitoring, assertion checking and verfication.

This paper discusses *CANDLE*, a high-level language and development environment, whose intended domain comprises embedded control systems in which computing nodes communicate using one or more Controller Area Networks (CAN). The *CANDLE* approach is novel in that it seeks to apply formal analysis to concrete implementations of distributed real-time systems, not only to their specifications. The essence of the approach is to provide a language and development environment in which a timed transition model of a system implementation can be produced almost as a by-product of a natural development method. The model can be used to explore system behaviour through simulation. In addition, a variety of abstractions can be applied in order to allow a tractable analysis of the model using model checking.

Keywords

Embedded Control Systems, Controller Area Network, Timed Transition Systems, Formal Modelling, Verification.

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 [12].

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 [8] have proved to be very successful models for the analysis of real-time systems [9] and they arise naturally from a variety of formalisms for system description, see for example [1, 6]. We argue in [12] 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.

The models which we produce are intended to allow the application of automated analysis techniques in order to investigate system properties. In particular, there is a straightforward translation of our models into formats which are suitable for input to the model checkers KRONOS [5] and UPPAAL [14]. With this in mind, we seek to build models which are both *accurate* enough so that we can be confident that the conclusions which we reach about a model are valid for the corresponding implementation, but also *abstract* enough so that analysis of the model is tractable. Our approach in this respect builds upon [3]. The main idea is to build a model which is a "conservative approximation" of an implementation. An abstract model of an implementation is a *conservative approximation* if every possible behaviour of the implementation is represented by some behaviour of the model, i.e. the behaviours of the implementation are a subset of the behaviours of the model. A formal demonstration of this relationship requires an approach such as that adopted in the ProCos project [13] where a system is viewed as a hierarchy of formally expressed semantic levels from

Proc. of the Fourth Workshop on Discrete Event Systems

the abstract (requirements and system architecture) to the concrete (switching circuits implemented in CMOS), in which each level can be related formally to its neighbours in the hierarchy. However, we are interested in short to medium term approaches to improving the quality of control systems developed with mass produced components. The availability of formal descriptions of such components is the exception rather than the rule. In which case, we proceed by a careful, but necessarily informal, development of conservative approximations, using data obtained from a variety of sources, including code timing analysis [15], simple scheduling analysis [2] and communication protocol standards [11].

Given an abstract model which is a conservative approximation of an implementation, we restrict attention to requirements which are expressed as properties of *all* behaviours of the model. In this case, 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 [4]. However, the work described there is restricted to single processor systems, whereas we are concerned primarily with distributed systems.

2 Informal control system model

Figure 1 shows a typical organisation for the class of control systems to be studied. Control is distributed over a number of *tasks* which are statically allocated to computing nodes. A computing node consists of at least a processing unit, which has access to some local memory, one or more communication controllers and a programmable timer. Tasks communicate by using one or more communication channels (buses) to send and receive broadcast messages. Each communication controller in a computing node is responsible for the node's access to a single channel. If a node communicates using more than one channel then it needs a communication controller for each channel that it uses. A number of tasks may be allocated to a single 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 passing 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. In the case of multi-tasking, it is assumed that sensors and actuators are not shared but that each is accessed by a single task.

We have targetted our development approach at a

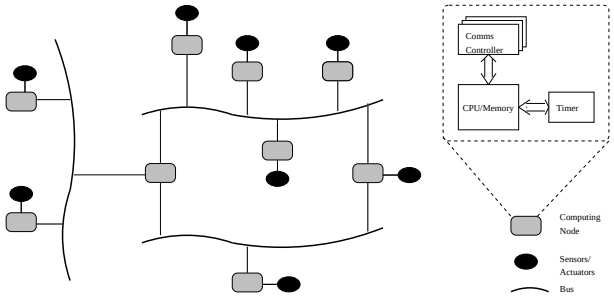

Figure 1: Control system model

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 [11].

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 [4, 10, 7]. Although only a simple system, this example allows the demonstration of nearly all 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 (`Sensor1.ReadSensor, Integrator.Signal` etc.). In general, it is assumed that sensors and actuators are not shared but that each is controlled directly by a single task; other tasks which desire access to a sensor/actuator must communicate their intentions to the controlling task by sending a CAN message.

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 uses *CANDLE* to describe the main details of implementations of one of the sensor tasks and the integrator task. The implementation of the other sensor task is similar to the one described.

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 Integrator`) establish a link to data specifications and implementations which are constructed using a suitable external language: we currently use B for specification and C for implementation. 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 by using the C code timing tool, `CINDERELLA` [15] in conjunction with a simple multitasking scheduling analysis as described in [2].

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].

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 [9], 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. The construction arises nat-

[1]What You Verify Is What You Execute

```
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 with data Integrator
  using
    constant PROXIMITY_MAX
    var sv1, sv2
    op Compute, Signal
  is
    loop do
      loop GATHER do
        select
           when rcv(sensor1,sv1) do
             select
               when rcv(sensor2,sv2) do
                 snd(ack1); snd(ack2); exit GATHER
             or
               when elapse PROXIMITY_MAX do skip
             end_select
        or
           when rcv(sensor2,sv2) do
             select
               when rcv(sensor1,sv1) do
                 snd(ack1); snd(ack2); exit GATHER
             or
               when elapse PROXIMITY_MAX do skip
             end_select
        end_select;
      end_loop GATHER
      Compute;
      Signal
    end_loop
  end_task
end_system
```

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

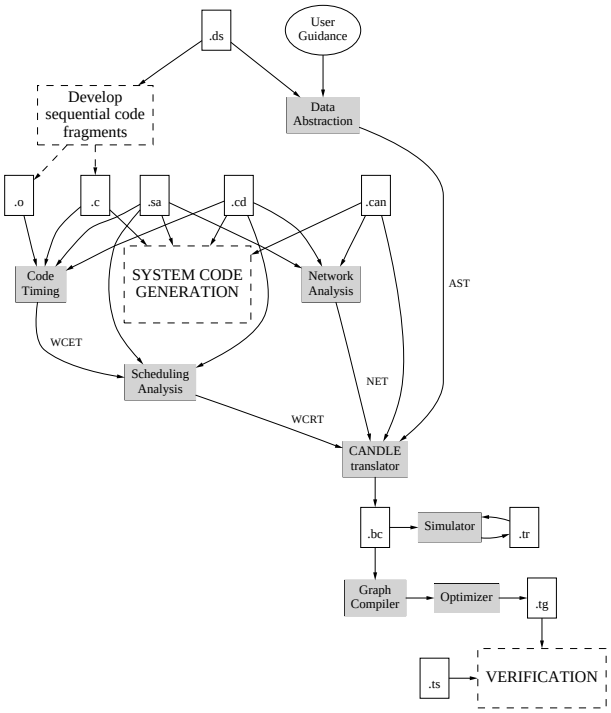

Figure 3: Architecture of the CANDLE development environment

urally from the use of the development environment shown in figure 3.

Figure 4 gives the *bCANDLE* model for the distributed 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, as with the construction of the network model, it relies upon information derived from a number of other sources. 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 execution. 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 multi-tasking node, the execution time bounds must be converted into response time bounds. This analysis is possible for a simple time-slicing scheduler [2]. In figure 4, 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

```
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 4: *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 [1]) acceptable to model checker or given by a description of a specification automaton.

Table 1: *CANDLE* development files

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

`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

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 and Further Work

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. The importance of quantitative timing analysis of such systems has been recognised for some time. Quantitative timing properties of systems depend directly upon their implementations. We aim to build systems in such a way that it is possible to extract abstract models which are conservative approximations of their implementations. These models are amenable to a variety of well-developed, tool-supported analysis techniques [9, 5, 14].

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.

A constraint on the class of systems to which our approach is applicable arises from the fact that our implementation of multi-tasking is currently limited to preemptive time-sliced scheduling, which allows an independent analysis to calculate the WCRT of all computations. Clearly there are some systems for which this may make it difficult occasionally to achieve the responsiveness that is required by some tasks; however, there are many for which this approach to scheduling is adequate. Future work will examine the extension of our model to accomodate fixed priority preemptive scheduling. Such an extension is possible but leads to harder model-checking and may constrain further the size of systems which can be analysed automatically. As usual, there is a trade-off to be made between the simplicity of the analysis and the sophistication of the implementation.

An essential requirement to allow the development and analysis of large-scale applications is for modular design, implementation and verification. *CANDLE*, in its full form, permits the composition of system descriptions, allowing channel sharing and the renaming of message identifiers where necessary. Inevitably, however, in general, composition does not preserve all system properties when the composed systems broadcast on a shared channel. We are currently investigating the use of rely-guarantee reasoning to allow the verification of some properties of compound systems based only on the construction and analysis of models of the components.

Finally, we wish to investigate to what extent it is possible and useful to develop an extended system model to reason about behaviour in the presence of network errors.

References

[1] R. Alur, C. Courcoubetis, and D. Dill. Automata for modelling real-time systems. In *Proc. 17th ICALP*, volume 443, pages 322–335, 1990.

[2] 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.

[3] S Bradley, W D Henderson, D Kendall, and A P Robson. Designing and implementing correct real-time systems. In H Langmaack, W-P de Roever, and J Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems FTRTFT '94, Lubeck, Lecture Notes in Computer Science 863*, pages 228–246. Springer-Verlag, September 1994.

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

[5] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In R. Alur, T. Henzinger, and E. Sontag, editors, *Proc. DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems (Hybrid Systems III)*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer Verlag, October 1995.

[6] 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.

[7] R. Gerber and I. Lee. A layered approach to automating the verification of real-time systems. *IEEE Transactions on Software Engineering*, 18(9):768–784, September 1992.

[8] T. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In J. de Bakker, C. Huizing, W-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop 'Real-Time: Theory in Practice'*, volume 600 of *Lecture Notes in Computer Science*, pages 226–251, Berlin, 1992. Springer Verlag.

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

[10] T. Henzinger and Pei-Hsin Ho. HyTech: The Cornell hybrid technology tool. In *Proc. 1994 Workshop on Hybrid Systems and Autonomous Control*, Lecture Notes in Computer Science, 1995.

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

[12] 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)*, pages 171–177. IEEE Computer Society Press, June 1998.

[13] H. Langmaack. The ProCos approach to correct systems. *Journal of Real-Time Systems*, 13:37–59, 1997.

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

[15] Y-T.S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of 16th IEEE Real-time Systems Symposium*, pages 298–307, 1995.