A Formal Basis for Tool-supported Simulation and Verification of Real-Time CAN Systems

D.Kendall, S.Bradley, W.D.Henderson, A.P.Robson

Abstract— **In this paper, we present a framework for the formal modelling of the temporal and functional behaviour of real-time distributed systems which communicate using one or more Controller Area Networks. A low-level modelling language is introduced whose timed transition semantics provides an abstract basis for the development of tools to support the simulation of industrial-strength systems and the verification of safety and liveness properties, including bounded response properties. The implementation and application of a simulator is described. We show how the integrated analysis of network and process behaviour permits a less pessimistic view to be taken of a wider range of system properties than is allowed by traditional scheduling analysis. The practical utility of the approach is emphasised and we illustrate how it can be applied to complex systems using a variety of CAN controllers and micro-controllers, including the 82527 and the MC68376.**

I. Introduction

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 central problem, as always, is to develop a 'system' to satisfy a given 'specification', taking all reasonable steps to demonstrate satisfaction. Testing has, and will continue to have, a major rôle to play in providing evidence of satisfaction. Post-hoc testing of arbitrary systems on its own, however, should not give developers great confidence in the continuing 'good behaviour' of their products; particularly when the product is a distributed, hard real-time, embedded control system. Simulation of a model of the system under development can give early reassurance that the development is proceeding along the right lines. Verification can provide even greater confidence. Formal methods have been successfully applied to the problem of verifying that abstract design models satisfy formal specifications of both functional and temporal properties. However, comparatively little work has sought to apply these techniques to implementation models. The state explosion problem is an effective deterrent. In this paper, we describe a modelling language which can be used to develop timed transition models of distributed control systems in which the processing elements communicate using a deterministic broadcast bus. We apply this approach to the industry standard Controller Area Network and show how the modelling language can be used to facilitate an integrated investigation of the behaviour both of computing tasks and of the network in CAN-based systems. The extent to which symbolic and modular verification techniques can be applied to mitigate the effects of the state explosion problem is an open question. Nevertheless, we have found that the approach provides a good basis for simulation, verification and implementation, and thus leads to increased confidence in the proper behaviour of systems.

The authors are with the Department of Computing, University of Northumbria at Newcastle, Ellison Place, Newcastle upon Tyne, NE1 8ST. Email:{david.kendall, steven.bradley, william.henderson, adrian.robson}@unn.ac.uk

The work described in this paper forms part of a programme whose objective is to provide a framework for the development of hard real-time distributed embedded systems, from requirements elicitation and validation to implementation. Our approach is guided by several criteria:

- We would like to realize a method which can be applied by system developers to problems of real interest. Application of the method should lead to justifiably increased confidence in the behaviour of delivered systems. Furthermore, developers should be free to choose standard components (hardware circuits, compilers, etc.) in the production of their chosen solutions, with the restriction that their run-time behaviour is *predictable*; in particular, it should be possible to put bounds upon the time taken to complete any run-time operation. Currently, this limits choice to *simple* components, although recent advances in execution time analysis are making it possible to reason about more complicated features [17].
- It should be possible to verify formally models of low-level implementations with respect to high-level specifications of requirements.
- We should be able to model and analyse both functional and temporal properties of systems.
- We wish to relate well-developed, stable theory from all phases of development – from requirements elicitation to scheduling theory – in a coherent approach, supported at each stage by appropriate software tools.

This paper introduces a framework which makes it possible to produce abstract models of distributed, real-time control systems which communicate using a deterministic broadcast communication protocol. Broadcast communication is used frequently in the implementation of distributed real-time systems, but has received comparatively little attention from the formal methods community in contrast to point-to-point synchronous communication. In the approach adopted here, we have attempted to maintain a clear separation of concerns in considering various aspects of implementation, particularly communication, concurrency, data and scheduling. This means that we can experiment with a variety of design and implementation techniques within the same overall framework.

The models which we produce are intended to allow the application of automated analysis techniques in order to investigate system properties. To achieve this goal, we need models which are *accurate* enough so that we can be confident that the conclusions which we reach about a model apply also to the implementation from which the model was developed, but also *abstract* enough so that analysis of the model is tractable. Our approach in this respect builds upon [5]. We adopt the idea of a "conservative abstraction" from [10]. An abstract model of an implementation is a *conservative abstraction* if every possible behaviour of the implementation is represented by some behaviour of the model. Although not desirable, a model may exhibit behaviours which do not correspond to any possible behaviour of the implementation which it models, i.e. the behaviours of the model are a superset of the behaviours of the implementation. This allows some necessary freedom in the development of an abstraction. By restricting attention to requirements which are expressed as properties of all behaviours, it is sufficient to establish that a model satisfies some requirement in order to conclude that the implementation from which the model was developed also satisfies the requirement.

The paper is organised as follows: section II introduces an informal system model; section III outlines properties of a CAN implementation which are assumed in the ensuing formal model. The low-level modelling language is introduced in section IV. Section V describes a simple manufacturing cell which is used as an example in sections VI, VII and VIII which discuss the construction of a low-level model, its simulation and verification, respectively. Section IX concludes and outlines our plans for further work.

II. Informal control system model

We address a class of control systems (see figure 1) which can be identified by a number of properties:

- 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.

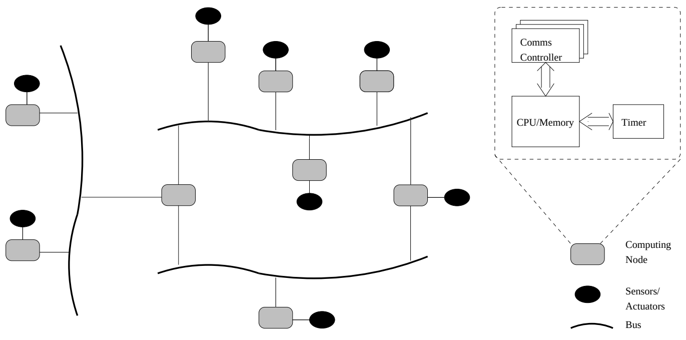

Fig. 1. Control system model

- 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 multitasking, it is assumed that sensors and actuators are not shared but that each is accessed by a single task.

III. Controller Area Network

CAN uses a simple, deterministic, broadcast communication protocol which makes it not only attractive to developers but also amenable to formal modelling and analysis. We assume that the basic principles of CAN are familiar and state here only those assumptions and simplifications which are relevant to what follows.

- A transmitting node attempts to transmit its highest priority message. (This requirement is satisfied trivially by the use of TouCAN controllers, for example, but requires more effort on the part of the programmer for a controller such as the 82527.)
- The transmitter with the frame of highest priority gains bus access *without experiencing any delay* due to any possible access conflict.
- A transmitting node which loses the arbitration recognises this fact and behaves as a receiver of the frame from that point on.
- A controller does not release the bus between transmissions, i.e. it enters a frame for arbitration in every arbitration phase if it has a frame to transmit. So lower priority frames cannot delay the transmission of pending frames of higher priority by beginning transmission during a gap between frames. The standard [14] p.26 says, 'A frame which is pending for transmission during the transmission of another frame is started in the first bit following intermission . . . '.
- It is guaranteed that a frame is "simultaneously" accepted either by all nodes which are configured to accept it, or by none of them (assuming that nodes are operating normally). There is no possibility of a "partially successful" transmission.
- We assume that we can determine the point during the transmission of a frame when a controller begins its acceptance test for the frame. In normal operation, a controller which becomes configured to accept messages at any time before it begins its acceptance test will accept all messages which pass the test thereafter.

• We ignore the possibility of error and overload frames.
• We ignore the distinction between error-active and error-passive nodes.
• We ignore the difference between the transmitter and the receiver of a message in determining the point at which a frame is valid (end of EOF for transmitter, end-of-EOF - 1 bit for receiver).

IV. LL: A low-level modelling language

The construction of an integrated model of a CAN-based system requires the modelling of both the static and dynamic properties of tasks, network and data environment in a single framework. This section introduces a low-level modelling language, LL, which is designed for this purpose.

A. Modelling the data environment

There are many well-known specification languages (Z, VDM and B, for example) which can be used to model data states and the effects of sequential operations upon them. Well defined programming languages are clearly suitable also. We want to allow the developer as much freedom as possible in their choice of approach. Whatever language is used, LL requires only that the following can be defined:

• A mapping, $D : Var \nrightarrow Value$, from variables to values which represents the data environment.
• A lookup operation $D.x$, which for any variable x denotes its value in the map D.
• An update operation $D[x := v]$, which denotes a data environment D', which is the same as D except that x is associated with v in D'.
• A binary relation $\overset{\omega}{\Longrightarrow}$, such that $D \overset{\omega}{\Longrightarrow} D'$ for each task computation ω, iff ω can be executed in D producing a new state D'. We use the operation label *ID* to represent the operation which leaves every data environment unchanged. $\forall D, D' \bullet D \overset{ID}{\Longrightarrow} D' \Leftrightarrow D = D'$.
• A binary relation, $\models$ between data environments and predicate symbols where for any D and any γ, $D \models \gamma$ iff the predicate associated with γ "holds" in D. We write $D \not\models \gamma$ for $\neg\ (D \models \gamma)$. We assume the existence of distinguished predicate symbols *true* and *false*, such that $\forall D \bullet D \models true$ and $\forall D \bullet D \not\models false$.

B. Modelling the network

The model of the network should allow us to answer a variety of questions for any given channel (CAN bus or internal data path):

• What messages are currently pending transmission?
• Is the channel currently in use or can a node begin the transmission of a new message?
• In the event that two or more nodes begin transmitting simultaneously, which will win the arbitration and succeed in transmitting its message?
• How long from the start of its transmission will it take for a message to become available for acceptance?
• By what point must a node be configured for acceptance of a message type in order to guarantee that it will acquire the next such message?

In order to give an answer to such questions, the network model must represent both the static and dynamic properties of each channel. Static properties of a channel are fixed throughout the execution of the system. They comprise:

• the name of the channel;
• the set of message identifiers which can be transmitted on the channel;
• a total ordering on the set of message identifiers according to their priority (a message μ_1 is of higher priority than a message μ_2, written $\mu_1 \sqsubseteq \mu_2$, if the message identifier of μ_1 is of higher priority than the message identifier of μ_2);
• a pair of functions δ^l_{pre} and δ^u_{pre}, from messages to time values, giving the lower and upper bounds on the time taken from the start of a message transmission to the point at which controllers begin their acceptance test for the message (the *pre-acceptance* phase of transmission); and
• a pair of functions δ^l_{post} and δ^u_{post}, from messages to time values, giving the lower and upper bounds on the time taken from the start of the acceptance tests to the channel becoming free for the next transmission (the *post-acceptance* phase of transmission).

The dynamic properties of a channel change as the system executes and are given by:

• the status of the channel (a channel is either *free*, in a *pre-acceptance* phase of transmission, at an *acceptance* point or in a *post-acceptance* phase of transmission);
• the priority-ordered queue of messages pending transmission on the channel.

The channels in a network can independently perform any of the following actions (except **Age Network**), changing thier dynamic properties accordingly:

Start Transmission When a channel is free, the highest priority pending message begins transmission and is removed from the pending message queue. If there are no pending messages, the channel simply idles, allowing time to pass.

Acceptance A channel which is transmitting a message, allows time to pass at most up to the latest time at which the controllers must begin their acceptance test for the message. The acceptance test can not begin before its earliest allowed time.

Finish Transmission When all properly configured nodes have performed their acceptance test for the message, the channel completes the transmission of any following bits in the message frame, time passing as it does so.

Release Channel When enough time has elapsed to complete the transmission of the message frame and to allow for the intermission, the channel becomes free again for the transmission of the next message.

Age Network Time can pass in a network only when it can do so for all of its channels.

C. Modelling the behaviour of tasks

The behaviour of tasks is modelled using a simple process language, given by the grammar:

$$\begin{aligned} P ::= {} & \kappa!\iota.x \mid \kappa?\iota.x \mid [\omega : t_1, t_2] \mid \gamma \texttt{ -> } P \\ \mid {} & P \texttt{ ; } Q \mid P + Q \mid P \, [> Q \mid P \mid Q \\ \mid {} & \mathrm{rec\,X.P} \mid \mathrm{X} \end{aligned}$$

where κ is a channel name, ι is a message identifier, x is a data variable, ω is a data operation, $t_1, t_2 \in \mathit{Time}_\infty$ are time values, γ is a predicate symbol, P, Q are process terms and X is a process variable. These terms represent basic processes which:

- enqueue a message for transmission on a channel,
- accept a message from a channel,
- perform a computation, and
- evaluate a guard;

and compound processes formed by:

- sequential composition,
- choice,
- interrupt, and
- parallel composition.

The operators bind from tightest to loosest according to the precedence ordering: `;`, `->`, `+`, `[>`, `|`.

Repetitive behaviour is modelled by recursion: `rec X.P`. The free variables of a term are those which are not bound by some recursion. The closed terms are those terms without free variables. We denote by $\mathit{Proc}_{\mathbf{Sys}}$ the set of closed terms which do not violate restrictions on the use of recursion: in particular, parallel composition is not allowed inside recursion and all use of recursion must be guarded by some non-zero time delay.

We use a number of syntactic abbreviations:

$$\begin{array}{cc} [t_1] \equiv [ID : t_1] & [\omega : t_1] \equiv [\omega : t_1, t_1] \\ [t_1, t_2] \equiv [ID : t_1, t_2] & \mathrm{skip} \equiv [0] \\ \mathrm{idle} \equiv \mathrm{false} \texttt{ -> } \mathrm{skip} & \end{array}$$

We also make use of equational definitions, exploiting the property that processes defined using a set of simultaneous equations have an equivalent description entirely in terms of the recursion operator.

Each process term, $P \in \mathit{Proc}_{\mathbf{Sys}}$, represents a potential process which, when given a control system context (i.e. a network and a data environment), is capable of exhibiting some behaviour.

Send The term, $\kappa!\iota.x$, denotes a process which causes a message to be queued for transmission on channel κ. The message consists of the message identifier, ι, and the data value associated with the variable, x. Sending is asynchronous. The process $\kappa!\iota.x$ can not be delayed. It causes its message to be queued instantaneously and terminates immediately.

Receive $\kappa?\iota.x$ is a process which waits to accept a message from channel κ. It will only accept a message with the identifier ι. It will ignore messages with any other identifier, simply allowing time to pass and other network activity to occur. When an ι-message reaches its acceptance point during transmission, then $\kappa?\iota.x$ must accept the message instantly, causing the data variable x to become associated with the message's data value. $\kappa?\iota.x$ then terminates immediately.

Compute $[\omega : t_1, t_2]$ is a process which transforms the data state according to the specification of the operation ω. It begins execution immediately and is guaranteed to terminate no later (resp. no sooner) than t_2 (resp. t_1) time units after it has started. The specified change to the data state occurs in a single, instantaneous action at the moment of termination.

Evaluate Guard γ `->` P causes the evaluation of the guard γ (which is a predicate on data states) in the current environment. If the guard is satisfied, the process P begins execution immediately; otherwise γ `->` P simply idles, allowing time to pass and network activity to occur.

Sequential Composition P ; Q behaves just as P until P terminates. It then carries on to behave as Q, using the state of the network and the data environment at P's termination.

Choice $P + Q$ behaves either as P or as Q. The choice is resolved in favour of whichever process can first perform an action. Network activity and the passage of time must be allowed by both P and Q in order to occur; neither resolves the choice. If both P and Q can perform an action simultaneously, the choice is resolved arbitrarily in favour of one of them.

Interrupt P [> Q behaves as P until either Q can perform an action or P terminates. In the first case, the system carries on to behave as Q with whatever is the current state of the network and data environment (P is aborted); in the second case, the whole process, P [> Q, terminates. Network activity and the passage of time both require the willingness of P and Q to allow them to occur. When time passes, it does so in both P and Q.

Parallel Composition The parallel operator, $P \mid Q$, gives a simple interleaving of the actions of P and Q. As with the other operators, network activity and the passage of time require the willingness of both P and Q to allow them to occur.

Recursion Equational definition gives the most readable expression of recursive processes. If $X \mathrel{\widehat{=}} P$ is a defining equation for X, an occurrence of the name X in a process term denotes a process which can behave as P.

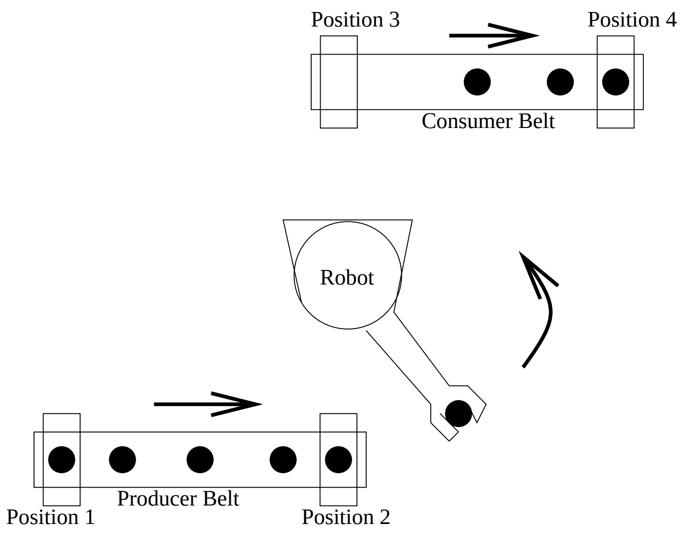

Fig. 2. Simple Manufacturing Cell

V. Example: A Simple Manufacturing Cell

Figure 2 illustrates a simple manufacturing cell which has been discussed in [6]. A production process, outside the cell, continually deposits items at position 1 of the producer conveyor belt. The producer belt controller drives the belt to move items from position 1 to position 2. The robot controller captures each item at position 2, rotates and processes the item. After processing, the robot rotates again and attempts to deposit the item on the consumer conveyor belt at position 3. The consumer belt controller operates the belt to move items from position 3 to position 4 where they are removed by some external consumption process. We outline an implementation in which the three controlling processes are physically distributed. They interact with the environment using sensors and actuators, and communicate with each other via a CAN.

Figures 3,4 and 5 show the main details of implementations of controllers for the producer belt, robot and consumer belt, respectively. This example informally introduces an Ada-like system description language, imaginatively called HL, which is more suitable than LL for describing complex broadcasting real-time systems. The language is given a formal semantics in terms of timed transition systems by translation into LL [15].

The manufacturing cell is implemented by three distributed tasks which execute in parallel and communicate via a CAN. They interact with the environment using a variety of sensors and actuators. Environmental interaction is modelled and implemented by simple sequential operations (`CheckPosition2, BeltOn, DepositItem` etc.) which are usually analysed independently to obtain bounds on performance. 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.

```
task ProducerBelt with data ProducerBelt
using
  constant PRODPERIOD
  var p1, p2, belton
  op CheckPosition1, CheckPosition2, BeltOn, BeltOff
is
  every PRODPERIOD do
    CheckPosition2;
    snd(pos2,p2);
    if p2 and belton then BeltOff end_if;
    CheckPosition1;
    if p1 and not (p2 or belton) then BeltOn end_if
  end_every
end_task
```

Fig. 3. Producer Belt Controller

```
task Robot with data Robot
using
  var p2, p3
  op ExtendArm, RetractArm,
     CaptureItem, DepositItem,
     RotateA_90, RotateC_180
is
  loop do
    rcv(pos2,p2);
    if p2 then
      ExtendArm; CaptureItem; RetractArm;
      RotateA_90;
      ExtendArm; ProcessItem; RetractArm;
      RotateA_90;
      loop RELEASE do
        rcv(pos3,p3);
        if not p3 then
          ExtendArm; DepositItem; RetractArm;
          exit RELEASE
        end_if
      end_loop;
      RotateC_180
    end_if
  end_loop
end_task
```

Fig. 4. Robot Controller

```
task ConsumerBelt with data ConsumerBelt
using
  constant CONSPERIOD
  var p3, p4, belton
  op CheckPosition3, CheckPosition4, BeltOn, BeltOff
is
  every CONSPERIOD do
    CheckPosition4;
    if p4 and belton then BeltOff end_if
    CheckPosition3;
    snd(pos3,p3);
    if p3 and not (p4 or belton) then BeltOn end_if
  end_every
end_task
```

Fig. 5. Consumer Belt Controller

The producer belt controller is a periodic task. It maintains three boolean variables `p1, p2` and `belton` to model the external environment. `p1` and `p2` are updated by `CheckPosition1` and `CheckPosition2`, respectively, and are set to `true` (resp. `false`) when the corresponding position sensor indicates the presence (resp. absence) of an item at its position. `belton` is updated by `BeltOn` and `BeltOff` in the expected way. Each period, the belt controller interacts with a belt sensor to check if an item has reached position 2. It broadcasts the sensor value on the CAN. If there is an item at position 2 and the belt is moving, then an actuator is triggered to turn the belt off. If an item is placed in position 1 when the belt is off and there is no item at position 2, the controller turns the belt on. The behaviour of the consumer belt controller is similar.

The robot controller is an event-driven task. Repeatedly, it waits to receive a CAN message, from the producer belt controller, which indicates the presence or absence of an item in position 2. When it detects the presence of an item, it actuates the robot to capture and process it, extending and retracting the robot arm and rotating the robot as necessary. Following processing, the robot controller seeks to deposit the processed item onto the consumer belt. It does so by repeatedly receiving CAN messages, from the consumer belt, regarding the status of position 3. When it receives a message indicating that there is no item at position 3, it causes the robot to deposit the processed item and to return to its original position.

The description of the CAN, by which the tasks communicate, is trivial and is not given here. It results in a single communication channel which carries two types of message, position 2 and position 3 status messages, distinguished by the message identifiers `pos2` and `pos3`. A task which wishes to communicate either by sending or receiving a message makes the necessary system call (`snd` or `rcv`) with parameters which denote the message identifier and the data variable for the message data value.

As discussed in section IV-A, the specification and implementation of the data environment for each task, and the operations which they can perform upon it, are expressed in a language of the developer's choice. We currently use B [1] for specification and C for implementation. Data clauses in `HL` (such as `with data ProducerBelt`) make the links to the appropriate specifications and implementations from which a description of the effect of the state transformers, and the bounds upon their performance, can be obtained.

VI. Constructing a low-level model

An `HL` system description must be translated into `LL` before its behaviour can be simulated or verified. We use the manufacturing cell example to introduce informally the translation and to illustrate salient points.

As discussed in section IV, an `LL` model is a triple (P, N, D) where P represents the control state of the system tasks, N represents the state of the network and D represents the data environment. Such a model is constructed automatically from the system description file, the data description files and auxiliary files describing the hardware configuration and kernel implementations. We can only sketch an outline of the construction here.

A. *D – Data Model*

The data model is simply a mapping from the task variable names to their values. Operations are modelled as relations between data states; a "before" state is related to an "after" state, by an operation, if execution of the operation in the "before" state is allowed and can lead to the "after" state. These state transforming relations are derived directly from the specification of the operations. The data spaces of the system tasks are required to be disjoint – no shared variables – so in the example, we use a prefix with each variable name to distinguish variables from different tasks: `P_` for producer belt variables; `R_` for robot variables; and `C_` for consumer belt variables. This gives an initial data environment for the manufacturing cell:

$$\{P\_p1 \mapsto \bot, P\_p2 \mapsto \bot, P\_belton \mapsto \bot,\\ R\_p2 \mapsto \bot, R\_p3 \mapsto \bot,\\ C\_p3 \mapsto \bot, C\_p4 \mapsto \bot, C\_belton \mapsto \bot\}$$

B. *N – Network Model*

The network model represents for each communication channel (a CAN bus or internal broadcast channel) its static attributes, status and pending message queue. Each channel is initially *free* and has an empty message queue. Its static attributes are constructed from a variety of sources:

- the *system description file* which, for each channel, defines its messages types, each having a distinct message identifier, and their priority ordering;
- *the data description files* from which can be obtained the size of communicated values; and
- *the hardware description files* which give details such as the channel's data transmission rate and the acceptance point for the controllers.

This information is sufficient to construct the static attributes for a channel. For a CAN bus operating at 1Mbit/s and serviced by i82527 controllers, we might construct the following table of attributes for the channel of the manufacturing cell[1]:

[1]In fact the network model for any system is augmented by a hidden "internal" channel for each task, so that a loop exit can be modelled as the sending of an internal message by which the task is waiting to be interrupted. Figure 6 shows an example of this use in the definition of `Robot`. The pre-acceptance time for the exit message is just the time taken for the task to execute a jump (post-acceptance time is zero), so the model remains a conservative abstraction of the implementation.

SA	`pos2._`	$\sqsubseteq$	`pos3._`
δ^l_{pre}	43		43
δ^u_{pre}	53		53
δ^l_{post}	10		10
δ^u_{post}	12		12

C. P – Task Model

Figure 6 shows the initial task state which is constructed for the manufacturing cell. This construction is based mainly upon the system description 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 [4]. In figure 6, 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, `[CheckPosition2]` represents a computation whose bounds are the calculated response time bounds for the operation `CheckPosition2`.
- 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 !]`, `[post !]` represent the bounds on the before and after delays for a `snd`, and `[pre ?]`, `[post ?]` 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 `ProducerBelt` and `ConsumerBelt`) requires similar information regarding their low-level implementation. We use `[pre Time]`, `[approx Time]` and `[post Time]` 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.
- `[eval guard]`, `[jump]` and `[branch]` denote the bounds that their names suggest. Occasionally, we have abbreviated a long sequence of computations and shown it as a single computation, for example `[CaptureProcessRotate]`; this is simply to help clarify the basic structure of the model.

VII. Simulation

A simulator for `LL` can be constructed directly from its formal semantics. A prototype simulator has been implemented by encoding the timed transition rules of `LL` as Prolog predicates. The system state is represented as a Prolog term and the rules can then be used to step through the possible behaviours. A production simulator would work directly with the underlying state transition graph. The user interacts with the simulator via a menu of choices which indicates which actions can be performed in the current state and how much time can elapse before an action transition must be taken. On choosing a transition, the resultant state can be used to investigate the condition of the network and the control and data states of the system tasks. The use of the simulator allows the system to be explored, providing the developer with an understanding of many of its properties before attempting to verify them. This can help to avoid much wasted effort in attempting to verify properties that the system does not possess.

VIII. Verification

Approaches to the automatic verification of finite-state concurrent systems have been known for more than a decade [12], [9]. Such techniques are based upon checking that the state graph of a concurrent system is a model for the temporal logic formulas which are used to specify desired system properties. Such an approach is of great practical interest because it allows the developer to verify a system without constructing a proof and because, when the verification fails, it is possible automatically to provide a trace of the unsatisfactory behaviour; this can be very useful in debugging and in fact such a trace can be used as input to a simulator so that the behaviour can be explored in detail. Recent work has shown how systems with a large number of states can be checked by using a symbolic representation of the state graph [7], [19] and how this approach can be adapted to the verification of real-time systems [3], [13].

A. Verifying requirements by model-checking

Research into the development of efficient model-checking techniques for real-time systems is currently very active [16]. KRONOS [20], [22], [21] is a symbolic model checker which implements the approach described by Henzinger et al. [13]. It allows timed transition systems to be checked for properties expressed in the real-time logic TCTL [2].

For a finite set of atomic propositions P, the formulas of TCTL are defined as follows:

$$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \, \exists\mathcal{U}_{\#n}\phi_2 \mid \phi_1 \, \forall\mathcal{U}_{\#n}\phi_2$$

where $p \in P$, n is a natural number and $\#$ is one of the relational operators $<, \leq, =, \geq$, or $>$.

TCTL formulas are interpreted over the sequences of states (paths) generated by a timed transition system. The details can be found in [13]. Intuitively, $\phi_1 \, \forall\mathcal{U}_{\#n}\phi_2$ means that every path has a finite prefix such that ϕ_2 is satisfied by the last state at time t where $t\#n$ and ϕ_1 is satisfied continuously until then. A number of abbreviations are commonly used, most importantly: $\forall\Diamond_{\#n}\phi$ for $\mathtt{true}\,\forall\mathcal{U}_{\#n}\phi$, which asserts that on all paths ϕ eventually holds within n time units; and $\forall\Box_{\#n}\phi$ for $\neg\exists\Diamond_{\#n}\neg\phi$, which says that

```
ProducerBelt | Robot | ConsumerBelt
where
ProducerBelt =
   [pre PRODPERIOD] ;
   ([CheckPosition2] ; [pre !] ; k!pos2.p2 ; [post !] ;
    [eval guard1] ; (guard1 -> [BeltOff] + not_guard1 -> [branch]) ;
    [CheckPosition1] ;
    [eval guard2] ; (guard2 -> [BeltOn] + not_guard2 -> [branch]) ; idle
   ) [> [approx PRODPERIOD] ; [post PRODEPERIOD] ; [jump] ; ProducerBelt

Robot =
  [pre ?] ; k?pos2.p2 ; [post ?] ;
  [eval guard3] ;
  (guard3 ->
     [CaptureProcessRotate]; (Release [> internal?RELEASE) ; [RotateC_180]
  + notguard3 -> [branch]) ; [jump] ; Robot

Release =
  [pre ?] ; k?pos3.p3 ; [post ?] ;
  [eval guard4] ;
  (guard4 -> [Deposit] ; internal!RELEASE ; idle + notguard4 -> [branch]) ;
  [jump] ; Release

ConsumerBelt =
  [pre CONSPERIOD] ;
  ([CheckPosition4] ; [eval guard5] ;
   (guard5 -> [BeltOff] + notguard5 -> [branch]) ;
   [CheckPosition3] ; [pre !] ; k!pos3.p3 ; [post !] ;
   [eval guard6] ; (guard6 -> [BeltOn] + not_guard6 -> [branch]) ; idle
  ) [> [approx CONSPERIOD]; [post CONSPERIOD] ; [jump] ; ConsumerBelt


guard1 == P_p2 and P_belton   guard2 == P_p1 and not (P_p2 or P_belton)
guard3 == R_p2                guard4 == not R_p3
guard5 == C_p4 and C_belton   guard6 == C_p3 and not (C_p4 or C_belton)

not_guardn == not guardn
```

Fig. 6. Manufacturing Cell: Initial Task State

on all paths ϕ holds continuously for n time units. The undecorated operators, $\forall\Diamond$ and $\forall\Box$ abbreviate $\forall\Diamond_{<\infty}$ and $\forall\Box_{<\infty}$, respectively.

TCTL is expressive enough to allow us to express most system properties of interest. For example, a bounded response property can be easily stated,

$$\forall\Box(\text{stimulus} \Rightarrow \forall\Diamond_{\leq 5}\text{response})$$

which captures the requirement that after any occurrence of a `stimulus`, a `response` will always happen within 5 time units. Other useful properties such as bounded invariance and bounded inevitability can be expressed just as easily.

The region construction of [2] extends the possibility of model-checking to systems with infinite state spaces caused by the use of a dense time model. However our state spaces are infinite because of the addition of infinite data types, most notably the use of an unbounded queue to represent the messages pending transmission on a CAN channel. We therefore need to apply data abstraction techniques [8], [18], [11] to all of our data types in order to produce finite representations which still preserve the properties in which we are interested. An obvious abstraction for a CAN channel queue, for example, is to maintain only the latest message to be queued for each distinct message identifier. Since the set of message identifiers for any channel is finite then the queues will be finite. This not only accords with most, if not all, existing implementations but also the application of the abstraction to the state space allows us to label those states, if any, in which duplicate message identifiers occur. It is then easy to check for such duplication, which may indicate an error condition such as the overwriting of an active transmission message object or the inadvertent use of the same message identifier by distinct nodes.

In the case of the manufacturing cell, there are many interesting properties which can be checked. Brockmeyer et al. [6] suggest: "Does the robot process ever enter the processing phase when there is no item in position" and "Does an item ever wait in position 2 for more than 20 time units before being processed". If we use proposional labels `P_P2` and `R_PROCESS` to label those states in which an item reaches position 2 and processing begins, respectively, then we can express the bounded response property in TCTL as:

$$\text{init} \Rightarrow \forall\Box\ \texttt{P\_P2} \Rightarrow \forall\Diamond_{<20}\ \texttt{R\_PROCESS}$$

Other properties can be tested similarly. For example, that the belt is turned off quickly enough when an item is detected at position 2; that the belt is not turned on while there is an item at position 2; that the robot begins processing only after capturing an item, and so on.

An advantage of model-checking is that a partial set of requirements, such as this, can be tested independently and failure traces obtained to aid debugging.

IX. Conclusions and Further Work

This paper describes ongoing work to provide a tool-supported engineering environment for the development of distributed embedded control systems. We believe that CAN will be an important component in many such systems because of its properties of robustness and predictability. We have therefore incorporated an abstraction of CAN's broadcast communication protocol into an integrated formal model of system behaviour. This model is being used as the basis of the development of tools for simulation, verification and code generation. The main advantage of such an approach is that it facilitates a fine-grained analysis of models which are abstractly related to the system implementation and which can be directly checked for compliance with both temporal and functional specifications.

So far, investigation has shown that implementations based upon the MC68376 and CPU+82527 are amenable to the approach described. The 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. HL, in its extended 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 a form of rely-guarantee reasoning to allow the verification of some properties of compound systems based only on the construction 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] J.R. Abrial. *The B Book – Assigning Programs to Meanings.* Cambridge University Press, 1996.

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

[3] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104:2 – 34, 1993.

[4] 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.

[5] S Bradley, W D Henderson, D Kendall, and A P Robson. Application-Oriented Real-Time Algebra. *Software Engineering Journal*, 9(5):201–212, September 1994.

[6] M. Brockmeyer, F. Jahanian, C. Heitmeyer, and B. Labaw. An approach to monitoring and assertion-checking of real-time specifications. In *Proceedings of the 4th IEEE Workshop in Parallel and Distributed Real-time Systems*, 1996.

[7] J.R. Burch, E.M. Clarke, K.L McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[8] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[9] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

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

[11] D. Dams, R. Gerth, and O. Grumberg. Absract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

[12] E.A. Emerson and E.M. Clarke. Using branching-time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

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

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

[15] D. Kendall, S. Bradley, W. Henderson, and A. Robson. A real-time formal model of broadcasting embedded control systems. Technical Report (to appear), University of Northumbria, 1997.

[16] K. Larsen, P. Pettersson, and Wang Yi. Model-checking for real-time systems. In *Proc. of Fundamentals of Computation Theory*, 1995.

[17] Y-T.S. Li, S. Malik, and A. Wolfe. Cache modelling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of 17th IEEE Real-time Systems Symposium*, pages 254–263, December 1996.

[18] C. Losieaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1995.

[19] K.L. McMillan. *Symbolic Model Checking: An approach to the State Explosion Problem.* Kluwer, 1993.

[20] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE Transactions of Software Engineering*, 18(9):794 – 804, 1992.

[21] A. Olivero and S. Yovine. *Kronos: A tool for verifying real-time systems – Users' guide and reference manual – draft 0.0*, 1993.

[22] S. Yovine. *Méthodes et Outils pour la Vérification Symbolique de Systèmes Temporisés.* PhD thesis, Institut National Polytechnique de Grenoble, May 1993.