Transputer architectures for sensing in a robot controller: formal methods for design

P.J. PROBERT, D. DJIAN AND HUOSHENG HU

Robotics Research Group
Department of Engineering Science
Oxford University
Oxford OX1 3PJ, UK

1. INTRODUCTION

This paper describes a sensing architecture being developed for our autonomous vehicle programme[4]. Each sensor uses a transputer as its local processor and will be either standalone or one in a network of similar or complementary devices. Such sensors exploit several features of the transputer: its bandwidth for data acquisition, its facilities for handling input/output and the ease with which networks of transputers can be built up. The latter is of particular importance in the context of sensor integration and an expanding system.

Designing robust and correct software is a particular difficulty in real-time, reactive systems. The problem is compounded by the emphasis on distribution and on synchronous communications to ensure a rapid response to external events. In our design, formal methods, even used at an elementary level, provided an essential tool for reasoning and analysis. An advantage of using transputers is that the transputer architecture and the language occam are directly related to the Communicating Sequential Protocols (CSP) language[3]. Process structure and communication follow the CSP model almost exactly; hence the formal structure of CSP may confidently be used to reason about the software design before it is implemented. This moves the main development effort into the design and specification stage and away from programming and debugging.

The architecture emphasizes distribution within a single sensor for robustness and more particularly speed.

1.1. Integration of sensory information

In the context of large real-time systems, key requirements for sensing are for reliability and high bandwidth. In mobile robotics typical needs are to assess range and geometric properties of the environment so that the robot can path-plan in a changing environment. Since information from a single sensor is noisy and incomplete, integration of data from a number of sensors is an important requirement.

Traditionally sensor integration has been done through the maintenance of a central world representation using a blackboard architecture[5]. However, the blackboard architecture has several problems, particularly related to bandwidth and the control of
system complexity. We have chosen a more structured architecture, in which integration is through either action or communication.

The architecture is shown in Figure 1. The vehicle is given a number of behavioural competences in the layered controller, following some of the ideas originally proposed by Brooks [1]. The principle of the architecture is that all layers operate concurrently with access to the actuators, but that the lower layers can subsume the operation of the higher ones as necessary; only one layer at any time actually controls the vehicle. Thus the vehicle may be controlled by the path-planning layer which plans actions through sophisticated sensing, but if the obstacle avoid layer sees an obstacle it takes control to guide the vehicle around that obstacle before allowing the higher layers to take control again.

Currently only the lowest level is operating under sensor control (a sonar ring), although outlines of higher levels are implemented.

![Layered Sensor Architecture Diagram](image1)

**Figure 1.** Layered architecture for control and sensing

**1.2. Sensor dependencies**

The architecture is designed for minimal interaction between layers. Sensor outputs may be available to more than one layer, but the sensory information in each layer is treated independently. (The lack of a global model may lead to difficulties if the individual models become too disparate; we are at present designing a formally proved architecture which includes loosely coupled communication between layers.) The only interaction is through the subsumption mechanism at the vehicle actuators, leading to integration of sensory information through the action of the vehicle.

Each layer of sensing is controlled by a transputer network which communicates to a transputer in the layered controller. Messages are sent between layers using transputer links.

Within each layer there may be interaction between several sensors which communicate
to establish a unified environmental model: integration through communication. The sensors in each layer are nodes of a communicating, distributed network. Distribution gives us advantages in bandwidth, robustness in the case of an individual processor failing and, with suitable design, expandability. Our sensor architecture maintains consistency within each layer whilst passing information on to the robot as it becomes available at each sensor (thus the speed is not reduced by a single slow sensor in a layer).

The remainder of this paper discusses the design of a sensor to fulfil these requirements.

2. COMMUNICATING SENSORS

As a case study, the immediate target of this work is a sensor which integrates sonar and infrared transducers to assess range, for use in the second level of our architecture. An embedded transputer provides local control and intelligence[2]. Sonar and infrared complement each other in their use, sonar providing good range information and infrared good angular resolution. The sensor differ too in bandwidth; readings from the sonar sensor are limited by the speed of sound and are expected to take several tens of milliseconds whereas the infrared sensor should respond in hundreds of nanoseconds (limited by the electronics).

The sonar and the infrared part of the sensor are implemented as separate logical sensors.

2.1. Overall specification

Although sensors vary in their physical characteristics each sensor must present the same interface. By working at a level of abstraction above implementation in terms of exact hardware and algorithms, we have developed a flexible model to handle all the sensors we expect to use in the mobile robot.

The functional specification of each sensor in the network is as follows.

1. The sensor should communicate with other sensors asynchronously. Asynchronous communication is preferred to exploit the bandwidth of the fastest sensor. In addition to communicating with other sensors each should send messages to an external party (in this case control information to the robot’s path planner) based on its own and other sensors’ observations (the world model).
2. The sensor design may be implemented on diverse processors, which may include special-purpose hardware. For rapid response to environmental changes internal communications are to be unbuffered where possible.
3. The sensor should include a facility to assess its own condition (for example to disassociate itself from the network if it believes itself to be faulty).

We refine this specification into the following logical processes.

1. *Hardware control*. Control of the transducer; acquisition of data. This function is the only one which requires access to physical hardware. It may be distributed between hardware and software.
2. *Data processing*. Local processing of the data, to establish assertions about the environment based on the sensor’s own readings.
3. Self-test. A self-modelling and -testing facility. The sensor should include some ability to determine whether it is working accurately.

4. Consensus. A module to provide communication with other sensors to maintain a consistent world model (in our architecture consistency is maintained only with other sensors in the same layer).

The division of the detailed functions into these general categories is governed largely by the requirement to keep communications between processes local and sparse. Design of suitable synchronization events became clear in the CSP description.

3. FORMAL DESCRIPTION

In CSP terms, a process is described in terms of the events in which it may engage. In occam an event is equivalent to a message passed along an channel. Each event requires the participation of two processes, one to send and one to receive. The synchronous communication model of CSP ensures successful process intercommunication.

3.1. Communications with the environment

The sensor may be described at many levels. At the highest level it is a single process, Lsensor, which communicates with the environment, the mobile robot and other sensors.

A model of the environment must be included as part of the specification. Consider first the interaction through the transducer (the emission and reflection of a sonar or infrared beam). The environment is assumed to be always prepared to participate in this process (in practice a time-out is conventionally provided to cover the case where there is no reflected beam detected). There is therefore no synchronization problem with the environment; it is always ready to send a signal. This is achieved in practice by appropriate programming of the C011 interface.

In addition the sensor interacts with the mobile robot. Although the sensors are viewed as a distributed network, interaction with the robot introduces a central function, which may need to communicate with a large number of diverse processes. The output from each sensor is information used for path planning. The complexity of the path planner varies considerably with the level of sophistication of each layer in the controller; at the lower levels control is largely reactive and path-planning is very simple. To cover a general case, and to avoid excessive complexity, the path-planner will be assumed always ready. (This condition can be approximated through a buffer process.)

Each sensor also needs information on the global frame of reference, possibly at the time it takes data but certainly to allow it to communicate its information in global co-ordinates. This information comes from the robot’s odometry. Again, because of the large number of sensors which may need this data, odometry information is assumed to be always ready.

The mechanism of communication with other sensors is defined within the sensor.

3.2. Formal specification of processes

Figure 2 shows the process model of the sensor, including the synchronization events between the processes. The processes are allowed to run concurrently, only limited by
the need for synchronization on the events shown. In a real implementation, some may be parallelized further.

The function of the sensor at any time is determined by its status, which is set by the Self\_test module as variable $s$. If $s = status.ok$ then the sensor takes its full part in the sensor network, passing its own readings onto other sensors. However, if the status is set to $s = partial\_failure$ the sensor enters a testing mode, in which it continues to get new data and process it (in the hope that the data received will lead to the sensor being deemed $status.ok$, possibly through a recalibration process within Self\_test) but does not broadcast this data. If the status is deemed to be $s = total\_failure$ the sensor disassociates itself from the network, but continues to accept messages from other sensors (necessary to avoid deadlock). Because the status controls the functionality, process Self\_test which determines it handles most of the co-ordinating and main control functions, communicating with all three other processes.

Hardware\_control represents the hardware interface to the transducer and any low-level data processing. On receiving event ready it initiates the event emit into the environment (i.e. the emission of a sonar/infrared beam). On receiving this beam back (event reflect) it transmits the data to Data\_processing and, if required, to Self\_test as well. So that the position information is up to date, the odometry information from the robot is available at this process, and information is transmitted in global co-ordinates.

Process Data\_processing takes in data data\_stream from the hardware control and processes it through a suitable algorithm, such as a Kalman filter. It also co-ordinates with the hardware through a control signal: get\_data. Processed data local\_assertion is sent to process Consensus, and to Self\_test.
Process Consensus has two functions. One is to hold and update a local version containing relevant features of a global model. The other is communication with the outside world. The global model is updated whenever a local assertion on the world, local.assertion or an assertion from another sensor, global.assertion is received. Since this process has access both to the local assertion and to global knowledge, it deduces the working condition of the sensor (i.e. whether self-testing is needed) and returns this information through message test.

The Appendix gives the process algebra of each process in CSP.

4. ANALYSIS OF DEADLOCK

4.1. A single sensor

The major difficulty in implementing and debugging concurrent systems is in detecting and removing conditions which may lead to deadlock. We would not have dared to design such a complex synchronization structure in the absence of tools for analysis—the proof rules of CSP[3]. Using their mathematical basis for predicting traces for processes in parallel we could ensure deadlock freedom (ensuring that no trace included the event STOP).

Since the proof is long (although tedious rather than difficult) it has not been included. The proof starts by combining together the relevant modules into the higher level process i.Interpretation (see the Appendix), where the prefix i denotes sensor i in the network. Then the processes i.Interpretation and i.Consensus are placed in parallel, synchronizing on events test and local.assertion to result in the algebra of a logical sensor, i.Lsensor. Hiding all internal events in i.Interpretation, except ready which shows when new data is acquired by the transducer:

\[
i.LSensor = (i.local.assertion \rightarrow i.j.global.assertion! \rightarrow i.test \rightarrow i.ready \rightarrow i.LSensor
| j.i.global.assertion? \rightarrow i.LSensor
| (s \leftarrow status.ok \rightarrow i.ready \rightarrow i.LSensor
| j.i.global.assertion? \rightarrow i.Lsensor)
| (s \leftarrow partial.failure \rightarrow j.i.global.assertion? \rightarrow i.Lsensor)
\]

Since the behaviour of process i.Lsensor does not contain the event STOP it is free from deadlock.

4.2. More than one sensor

A second sensor j.Lsensor is identical to the expression above but with the suffixes i and j interchanged). Consider now the combination of sensors: i.Lsensor || j.Lsensor.

Although each sensor is deadlock-free, it can be seen by inspection or from using the CSP proof rules that the combination may enter deadlock through both sensors simultaneously entering the symmetrical upper guards (i.e. receiving signals i.local.assertion and j.local.assertion respectively). Each will wait for synchronization from the other on the signal global.assertion before recursing.
This problem results from allowing the sensors to operate asynchronously. There are in fact often deadlock problems if two symmetrical processes are placed in parallel. There are two ways around this: either to introduce buffering at this level or to introduce an additional process to schedule the two sensors using a synchronizing signal. We investigated the latter option to maintain the emphasis on message-based communication.

4.3. A scheduler

Figure 3 shows a solution to the problem. The scheduler is placed between the processes Consensus and Interpretation of each sensor, ensuring that only one sensor at a time can enter its first guard, transmitting information $i.info$. Consensus then becomes:

$$i.Consensus = (i.info \rightarrow i.j.global.assertion! \rightarrow i.test! \rightarrow i.release! \rightarrow i.Consensus$$
$$\mid j.i.global.assertion? \rightarrow i.Consensus)$$

where events $i.info$ replaces $i.local.assertion$ and event $i.release$ denotes the end of the transmission of the sensor's map to the other sensor.

![Diagram of the scheduler process](image)

*Figure 3. The scheduler process*

The scheduler process synchronizes with $i.Interpretation$ on the local data $i.local.assertion$, ensuring that both processes Consensus cannot enter the upper guard simultaneously:

$$Scheduler = (i.local.assertion? \rightarrow i.info! \rightarrow i.release? \rightarrow Scheduler$$
$$\mid j.local.assertion? \rightarrow j.info! \rightarrow j.release? \rightarrow Scheduler)$$
Now if a \textit{i.Consensus} enters the upper guard through receiving \textit{i.info}, the process \textit{Scheduler} ensures that \textit{j.Consensus} does not enter its upper guard (through receiving \textit{j.info}) until \textit{i.j.global.assertion} is available. This resolves the deadlock problem. A formal proof of deadlock freedom for the two sensors may be found in [6]. The scheduler should expand to handle more than two sensors without difficulty.

The inclusion of the scheduler introduces a central process into the distributed network at a level before the sensor integration. However, most of the sensor functionality still operates entirely independently of other sensors and so, although we had hoped to leave the centralization until the robot itself, the scheduler does not significantly detract from the distributed nature of the sensing. Buffering must be introduced if such a dependence is not acceptable.

5. FROM CSP TO IMPLEMENTATION

One of the claims of formal languages is that they are implementation-independent. This is both an advantage and a potential problem, since there is not necessarily a hardware or software facility to realize their abstractions. However, the relationship between CSP and occam is so close as to provide an almost automatic coding mechanism.

The CSP specification makes no assumptions on the mapping of process to processor. This allows us to write general and versatile specifications. In implementation we may want to use different mapping for different sensors, depending on their complexity: in particular sensors with computationally intensive data processing may use a transputer (or even a network) dedicated to the processing algorithm.

The two sensors under immediate consideration, the sonar and infrared, may be controlled by a single transputer or have a transputer built into each. With a dedicated transputer, three links are in use. One goes to the transducer through the hardware control, one to the other sensor and one to the outside world, the mobile robot. The fourth link is left free so that more sensors can be added. However the structure supports a flexible implementation.

In a real implementation some processes will probably run on a single transputer—in pseudo-concurrence controlled by the microcoded scheduler in the transputer. The deadlock freedom proved in the CSP holds for all scheduling algorithms (although it should be noted that CSP has no notion of fairness).

6. CONCLUSIONS

We have demonstrated an architecture for sensor integration, using action between layers in a subsumption-type architecture and intercommunication within a layer. An architecture for a robust, intelligent sensor has been proved to be deadlock-free using CSP rules for two such asynchronous intercommunicating sensors.

In this analysis we have paid no attention to timing. Formal techniques which include timing information are becoming available but they are still difficult to use. A case study of the mobile robot controller using timed CSP is being developed [7].

The availability of a higher-level mathematical notation enforced a specification discipline and offered a simple method to expose and interpret the important aspects of processes, especially formulating inter-process synchronization. The proof rules allowed
us to ensure the absence of deadlock in a simple mechanistic way and successfully to
design a distributed unbuffered architecture for real-time control applications.

ACKNOWLEDGEMENTS
We thank Jeff Sanders, of the Programming Research Group, for his help in formulating
the CSP descriptions.
This work is funded by SERC.

REFERENCES
2. H.F. Durrant-Whyte, S. Grime and H. Hu, 'A modular, decentralised architecture for multi-
4. M.D. Adams, H. Hu and P.J. Probert, "Towards a real time architecture for obstacle avoidance
   and path planning in mobile robotics", Proc. IEEE Conference on Robotics and Automation,
   Cincinnati, 1990, pp. 584–90.
5. H.P. Nii, 'Blackboard systems: the blackboard model of problem solving and the evolution of
   blackboard architectures', AI Magazine, 7(2) 38–53, 7(3) 82–106 (1986).
6. P.J. Probert and D. Djian, 'Case study in formal specification for sensor integration', Technical

APPENDIX: CSP Process Algebras
This Appendix gives the CSP definitions for each process, hiding internal events.

\[ \text{Self.test} = \text{status!s} \rightarrow \text{SKIP}; \]
\[ \text{(procdato? } \rightarrow \text{test?x } \rightarrow \text{ready!x } \rightarrow \]
\[ \text{(test\_data? } \rightarrow \text{Self.test}\]
\[ \text{< x=\text{self.test.req} \triangleright } \]
\[ \text{Self.test}\]
\[ \text{< s=\text{status.ok} \triangleright } \]
\[ \text{ready!next } \rightarrow \text{Self.test}\]
\[ \text{< s=\text{partial\_failure} \triangleright } \]
\[ \text{Self.test}\]

where

1. variable \( x \) has value \( x=\text{self.test.req} \) if the sensor should perform a self-test to check
   its status and \( x=\text{next} \) otherwise
2. the construct \( \text{< x = self.test.req> } \) is interpreted to mean that the action above is
   implemented if the condition within the construct is \text{true}.

Including the environmental synchronization on events \text{emit} and \text{reflect}:
Hardware\_control = emit → reflect → datastream! → ready?x →
(test\_data! → Hardware\_control
<x=self.test\_req >
Hardware\_control)

Data\_processing = status?s → datastream? → procdata! →
(local\_assertion! → Data\_Processing
<s=status.ok >
Data\_Processing)

The higher-level module Interpretation is the combination of Hardware\_control, Data\_processing and Self\_test. Communication between Interpretation and Consensus is through the events test and the local sensor interpretation local\_assertion. Hiding all internal events except ready, to show when new sensor data is acquired:

\[ \text{Interpretation} = \text{Hardware\_control} \parallel \text{Data\_processing} \parallel \text{Self\_test} \]
\[ =\text{local\_assertion! → test? → ready.x → Interpretation} \]
\[ < s=status.ok > \]
\[ \text{ready.next → Interpretation} \]
\[ < s=partial.failure > \]
\[ \text{Interpretation} \]

Denoting the sensors by the prefixes \( i \) and \( j \), for sensor \( i \):

\[ i.\text{Consensus} = \]
\[ i.\text{local\_assertion}? → i.j.\text{global\_assertion!} → i.\text{test!x} → i.\text{Consensus} \]
\[ | j.i.\text{global\_assertion}? → i.\text{Consensus} \]

The definition of the logical sensor, \( L\text{sensor} \) is the parallel combination of Interpretation and Consensus.