Design of an Embedded System using Sequential Communicating Processes

Implementation of LEGO Car with XMOS

Fan Xiao
Ohmori Lab
CIS, Hosei University
Tokyo, Japan
xiao.fan.6b@stu.hosei.ac.jp

Abstract—As demands evolve, the concurrent technology becomes more and more important in embedded systems. Concurrency is to make full use of every core on processors to achieve maximum processing performance. When dealing with concurrent processes, these processes simultaneously access the same resource, they may conflict with each other, so we need concurrency control. As secure concurrency control design is not easy, embedded systems commonly use a round-robin to achieve multi-task, in this study, an embedded system designed by concurrent processes will be implemented from a completely different perspective, which is to propose a new development method and evaluate it by implementing a sophisticated system, which is realized by a LEGO car.

Keywords—embedded systems, communicating sequential processes, concurrent, XMOS

I. INTRODUCTION

With the development of information technology, intelligent and network, embedded systems are becoming increasingly widely used, and high processing performance of embedded systems is necessary. Usually, the embedded systems we are talking about are composed of microprocessors or microcontrollers which are pre-programmed to perform multiple tasks. The hardware resources of embedded systems are limited, which requires each core of the processor to be fully utilized to achieve maximum processing performance. Therefore, in embedded systems, concurrent technology is getting more important.

In a concurrent process of an embedded system, when concurrent processes simultaneously access the same resource, they may conflict with each other, which will lead to competition adventure, deadlocks, resource depletion and other problems, so the design of stability and effective security concurrent control is very necessary.

There exists a few concurrent control technologies to achieve multi processes. For instance, time slice allocation mechanism, round-robin algorithm, using interrupt to switch tasks, etc.

The conventional process allocation use interruptions, this causes skew since an urgent process, which if has to be handled immediately, must wait until a CPU is allocated. It is not good for an embedded system which has a high requirement on real-time.

Besides, as events caused by inputs and outputs are handled in a complicated way, a conventional operating system provides a software-driver for each peripheral type. This technology makes it impossible for the user to install his own peripheral to the system without knowledge related to software-drives. It is also not good for an embedded system.

To deal with these problems, a new development method, the combination of CSP [1] and XMOS, is proposed in this paper and evaluated by using implementing a sophisticated system – a LEGO car [2].

We use CSP to model the embedded system designed by concurrent processes, then realize it on the LEGO car built by XMOS processor and LEGO components.

This work can be divided into three main parts:

- CSP description of the embedded system
- CSP model simulation and verification using PAT3
- CSP model realization on a LEGO car built using XMOS processor and LEGO components

Task requirements of the embedded system (the LEGO car):

- Functional requirements:
  1. Line-tracing: tracing the black line
  2. Crossroad-detect: turn to the left road when come to the crossroad
  3. Color-distinguish: to distinguish different color balls and make corresponding respond
  4. Obstacle-avoidance: turn around when there exits obstacle on the road

- Performance requirements:
  1. Input voltage: motor drive voltage is 9V, the power supply voltage of XMOS processor and sensors are 5v.
  2. Fluency and Stability
II. DESIGNING SCHEME

A. Technical Route

- Specification stage: In this part, we consider about what kinds of car we want to design, and how the car behaves. Then we specify the car system.
- Assembling stage: Assemble the LEGO car and prepare related hardware.
- CSP stage: Build the embedded system model based on the task requirements.
- PAT stage: Describe the CSP model in PAT, simulate the system, generate state transition diagram and perform a few checkings to make sure the system is reliable without deadlock, livelock, etc.
- XC stage: Transform the system model from formal language to procedure-oriented language (XC [3]) which can be directly downloaded to XMOS processor.
- Run stage: Run the XC-described system on the LEGO car.

B. Key Technology

- CSP

Communication Sequential Process (CSP) is a mathematical theory known as process algebras developed by C.A.R. Hoare to solve concurrent phenomenon. It is an abstract language to describe physical behavior of message-passing communication in concurrent systems.

CSP provides two classes of primitives in its process algebra: Events and Primitive processes. Events represent communication or interaction between processes, which are indivisible and instantaneous, they may be atomic names (e.g. `on, off`), compound names (e.g. `door.open, door.close`), or input/output events (e.g. `keyboard? x, screen! bitmap`). Primitive processes represent fundamental behaviors.

CSP uses alphabet, traces and refusals to describe a process. Use $aP$ to denote a collection of all the events that the process $P$ (including other related processes) may contain. Use $traces(P)$ to describe a finite sequence of symbols recording the events in which the process has engaged up to some moment in time. A failure process is described by a set of $(s, X)$ which is the combination of traces and refusals. The first element $s$ is the traces of the process, the second element $X$ is the refusals of the process based on traces $s$, it is a set of events the process can refuse to perform. If a process can refuse to execute any event, the process is in a deadlock.

Process runs like this: select any initial action to perform, wait for the end state of this action, select an action from this end state, another wait, and another select... That may require more than one independent process to perform an action, in which situation inter-process communication is needed. A communication is an event that is described by a pair $c,v$ where $c$ is the name of the channel on which the communication takes place and $v$ is the value of the message which passes. A channel provides a synchronous, point-to-point connection between two threads over which messages may be communicated. The set of all messages which $P$ can communicate on channel $c$ is defined $ac(P) = \{ v \mid c.v \in aP \}$. Two functions are defined to extract channel and message components of a communication: $channel(c.v) = c$, $message(c.v) = v$, they can separate communication channel and message out: $channel(cv) = c$, $message(cv) = v$. So a process which is initially prepared to input any value $v$ communicable on the channel $c$, and then behave like $P(x)$, is defined $(c?x \rightarrow P(x)) = (y : \{ y | channel(y) = c \} \rightarrow P(message(y)))$.

- PAT

The Process Analysis Toolkit (PAT) is a CSP analysis tool. PAT is able to perform refinement checking, LTL model-checking, and simulation of CSP and Timed CSP processes. The PAT process language extends CSP with support for mutable shared variables, asynchronous message passing, and a variety of fairness and quantitative time related process constructs.

- Incrementally Modular Abstraction Hierarchy (IMAH)

The innumerable combination of agents causes combinatorial explosion and logical faults which we should avoid. Here we use IMAH [4-6] method to avoid such problems. IMAH is a common method that can be applied to wide variety of application areas. IMAH is divided into two parts: abstract part and concrete part. In the abstract part, the sequences of events and state transition diagrams are designed. In the concrete part, communicating sequential processes and programming codes are implemented. These two parts can be described more precisely in seven abstraction levels: The abstract part includes five levels from the homotopy level to the cellular space level. The concrete part consists of the presentation level and the view level. The abstract part generates common models in hardware and software. The concrete part creates specific models of hardware and software.

- XMOS

XMOS launched the event-driven xCORE, which is a kind of multicore microcontroller that delivers scalable, parallel multitasking computing. It can be configured to support a wide range of interfaces and peripherals, and responds much faster than conventional microcontrollers to deliver precise real-time performance.

A traditional processor like ARM realizes multi processes by time-slicing in a preemptive multi-tasking system, in which inputs and outputs cause interruption. Actually, only one process runs at a time. On the other hand, an XMOS processor truly runs multi-processes concurrently. Inputs and outputs of an XMOS processor are treated as events. By combining multiple XMOS processors, it is possible to execute a huge number of processes concurrently.

Fig. 1 shows the timing diagram of dealing with two concurrently running tasks (TaskA and TaskB) using both traditional CPU and XMOS. The traditional CPU uses interruption to realize processing concurrency, while in XMOS there's no interruption, it uses events.

---

- 16 -
I. **CONCRETE IMPLEMENTATION**

A. **Hardware Design and Implementation**

We built the LEGO car based on the LEGO NXT robot, which is rich in scalability. It provides an environment of developing a robot using LEGO components and a programmable computer known as the NXT Intelligent Brick, installing an ARM processor. Instead of using the Intelligent Brick, we propose to use XK-I processor [7]. We use XK-I to deal with three types of threads. The first type of threads receives inputs from input devices such as sensors. The second type of threads sends outputs from output devices such as motors. The third type of threads controls the system by receiving messages from a thread of the first type and sending messages to another thread of the second type.

According to the functions, the car can be divided into the following six modules:

1. **Power Module**: we use a battery box to provide a voltage of 9V to the SuperPro Board which can transform 9V to 3.3V and 5V, so we got 9V, 5V and 3.3V.

2. **Motor Module**: this module consists of two front wheels controlled by two independent servo motors and one universal wheel in the back; the two servo motors are controlled by a Motor Multiplexer through one I²C interface.

3. **Obstacle-avoidance module**: this module has an ultrasonic sensor to detect obstacle in the road, once there exists obstacle in the road, the car turns around for 180 degrees and goes back, here we use line sensor to make sure that the car can continue trace black line after turning.

4. **Tracing Module**: this module has a line sensor to trace black line and tell crossroad;

5. **Color-distinguish Module**: this module uses color sensor to distinguish different color balls.

6. **Other Module**: this module is used to test a few analog sensors which are connected through SuperPro Board.

The whole structure of the LEGO car is shown in Fig. 3:
Using FC, an agent produced a command to its device. The command is a notification by sending data from the agent to the device or an order of having the device send data to the agent. As commands and data are transmitted via an FC digital interface, we can easily connect devices to the car once they understand its principle.

Some devices are equipped with analog interfaces. In this case, these devices are connected through SuperPro Board which has analog-to-digital converter.

B. Software Design and Implementation

After assembling LEGO blocks, designing and implementing the hardware system, it comes to the design and implementation the software system. This software system is designed using Incrementally Modular Abstraction Hierarchy (IMAH) while avoiding logical faults. IMAH is a common method that can be applied to wide variety of application areas. Using IMAH, the LEGO car is carried out by descending the abstraction hierarchy, where the specification is transformed to components, state transition diagram, the description of communicating sequential processes and program codes.

- Applying the IMAH Method

The software system of the LEGO car has seven services: controlling the car, detecting the black line, detecting the crossroad, detecting obstacles, detecting color balls, rotating the left wheel and rotating the right wheel. Among them, the service for controlling the car is considered as the most important one.

The homotopy level of the hardware design is achieved by providing 7 connected spaces as shown in Fig. 5, each of which stands for the processor or a peripheral device. The connection of spaces is not decided until some concrete level design is finished. The homotopy level design for software provides 6 independent agents, each of which takes care of the allocated peripheral device. These agents are also shown in Fig. 5.

Fig. 5. Spaces and Agents

Then we come to the set theoretical level; here a software agent or hardware component is more concreted by providing services or functions that it provides. A set is accommodated to each agent. Clarified services become an element of the set. Take Ultrasonic Agent for example. The ultrasonic sensor has services to the ultrasonic sensor and to send it to the Motor agent. The general set for this agent is \{Ultrasonic, Motor\}. This can be more detail described as Ultrasonic Agent = \{ normal-state, obstacle-state, on-black-line, off-black-line, obstacle, non-obstacle, rotate-right-wheel, inverse-rotate-right-wheel\}.

At the topological space level, the relation among services or functions is clarified. The sequences of services can be represented as a graph in such a way a service becomes a vertex and a transition from one service to another becomes an edge. So the topological graph of the Ultrasonic Agent can be describe as shown in Fig. 6. In particular, the sequence of events in services is determined.

Fig. 6. A sequences of services represented using a topological graph

In the same way, other agents can be depicted.

At the adjoining space level, agents are attached together, here we use Ultrasonic Agents and Line Agents to display which is shown in Fig. 7. The two separated spaces that share part of them are combined in one space by attaching sharing spaces. The two tasks share the states normal-state, which are combined into one space. At the states normal-state, one task has a cyclic event no-barrier and the other task does on-black-line, these events are combined as an event no-barrier ∩ on-black-line.

Fig. 7. The two Agents are attached together

At the cellular space level, a Mealy machine, which is a state transition diagram, is transformed from the sequence of events is obtained as shown in Fig. 8. Each service is represented by a Mealy machine so that it is possible to implement the service not only by means of hardware but also by means of software. In case of hardware, each service becomes an independent sequential logical circuit. As services interact with each other, their sequential logical circuits also communicate each other. In case of software, the services become processes (or threads), which communicate each other.
At the presentation level, processes are obtained from the state transition diagrams.

At the view level, processes are transformed into programing codes, that presented in the following CSP & XC part.

- CSP & XC

Let's see the CSP code of the main function (more is analysed in the thesis):

```csp
// the sensor control class
Control() = Line() || Color() || Ultrasonic();
// the motor movement class
Movement() = Track() [*] TurnAround() [*] ColorDetect();
// System
System() = Control() || Movement();
SystemQ is the whole car system process. The Control() process and Movement() process are working on different threads, they communicate with each other through channels. Actually four processes are working concurrently on four threads in this system: three processes in Control() and the Movement().

After using PATS to simulate and verify the CSP model, we're ready to transform the CSP model to XC language. The concrete details will be shown in Verification and Test part.

The main function of the system in XC shown below is really compact with a single par construct, the advantage using XC is very obvious.

```csp
int main(void) {
  chan la, lr, c, u, s; // channel definition
  par
    Line(la, lr);
    Ultrasonic(u);
    Color(c, s);
    Motor(la, lr, c, u, s);
  } return 0;
}
```

In the main function, we created four concurrent threads, all of which run separate tasks independently of one another; the first three threads communicate with the motor thread through five channels.

Here we use the Color-Distinguish Module as an example to show the details of transform from CSP to XC.

First, the ColorDetect() process get color data from channel c. Here we considered two colors: red and blue with the color value 9 and 2 from the color sensor. We use if condition to make choice. Fig. 9 shows the details of the transforming process from CSP to XC.

![Fig. 8. The Meely machine, which is a state transition diagram](image)

![Fig. 9. The program codes are obtained from CSP](image)

- I²C protocol

The I²C protocol is consist of the following parts:

1. Data validity
2. START and STOP conditions
3. Transferring data
4. Data formats with 7-bit addresses
5. 7-bit addressing

I2C struct definition in XC (defined in i2c.h):

```c
struct r_i2c {
  port scl; // Port on which clock wire is attached.
  port sda; // Port on which data wire is attached.
  unsigned int clockTicks; /* Number of reference clocks per I2C clock, set to 1000 for 100 Khz*/
};
```
Function to write data to slave devices (defined in i2c.h) is shown as follows:

```c
int i2c_master_write_reg(
    int device, // Bus address of device, even number between 0x00 and 0xFE.
    int reg_addr, // Address of register to write to, value between 0x00 and 0x7F.
    unsigned char data[], // Array where data is stored.
    int nbytes, // Number of bytes to read and store in data.
    REFERENCE_PARAM(struct r_i2c, i2c_master) // I2C struct containing the clock and data ports.
);
```

Function to read data from slave devices (defined in i2c.h) is shown as follows:

```c
int i2c_master_read_reg(
    int device, // Bus address of device, even number between 0x00 and 0xFE.
    int reg_addr, // Address of register to read from, value between 0x00 and 0x7F.
    unsigned char data[], // Array where data is stored.
    int nbytes, // Number of bytes to read and store in data.
    REFERENCE_PARAM(struct r_i2c, i2c_master) // I2C struct containing the clock and data ports.
);
```

IV. VERIFICATION AND TEST

A. CSP Model Verification

The verification of the software system is carried out before performance test using PAT3.

We simulate the car system model and generate the state transition diagram shown in Fig. 10.

![State transition diagram generated by PAT3](attachment:image)

And through verification, we know the system is deterministic and deadlock-free, the result is shown in Fig. 11.

![Verification result](attachment:image)

B. LEGO Car Test

- Performance test
  - Performance of Motor Module, Obstacle-avoidance module, Tracing Module, Color-distinguish Module is all fine. They run fluently with few mistakes. The analogy sensors in the Other module part do not work good because of the noise effect of the motors. The Lego car is shown in Fig. 12.

![The LEGO car](attachment:image)

- Test Video.

V. CONCLUSION

The LEGO car has been successfully designed and implemented. CSP and XMOS made it possible to design and implement the hardware and software easily since each service was realized as an individual thread and an event was performed as message passing.

In the future work, we would try to calculate the performance advantage of XMOS processor over the traditional processor (like ARM) when dealing with concurrent processes in some certain situation.

REFERENCES