Communication Model Exploration for Distributed Embedded Systems and System Level Interpretations

Takashi Kinoshima, Kazutaka Kobayashi, Nurul Azma Zakaria, Masahiro Kimura, Noriko Matsumoto, and Norihiko Yoshida

Division of Mathematics, Electronics and Informatics
Saitama University, Saitama 338-8570, Japan
yoshida@ics.saitama-u.ac.jp

Abstract. This paper presents how communication exploration can be done in a design process of distributed embedded systems. Distributed embedded systems involve various communication categories such as event-triggered and time-triggered communication. Therefore, communication exploration is as important as architecture exploration. A design process begins from abstract specification without assuming any communication category, then explores the categories in a stepwise manner, and is followed by physical implementation synthesis. This paper includes system level interpretation of the communication models using the SpecC language so as to verify them.

Keywords: Distributed Embedded Systems, Event-Triggered Communication, Time-Triggered Communication, Stepwise Refinement Design, Model-Driven Architecture.

1 Introduction

Modern embedded systems often work in networks, which comprise *distributed embedded systems*, as found in vehicles for example. Distributed embedded systems involve communication in various layers, from bus connections to networks, thus communication design is more important and difficult than in single embedded systems.

System-Level Design have been gradually used into practice for embedded system design. Its typical design process proceeds as shown in Fig. 1 [1,2]. It is a stepwise refinement process from abstract specification to implementation.

An issue, from the network point of view, in the above process when applied to design of distributed embedded systems is that communication is concerned mainly with bus connections, thus communication exploration is not separated from architecture exploration, in which an suitable combination of modules is explored among several possibilities to fix an architecture model [3].

This paper presents how communication exploration can be done in a design process of distributed embedded systems using an example of event-triggered

M. Denko et al. (Eds.): EUC Workshops 2007, LNCS 4809, pp. 355–364, 2007.

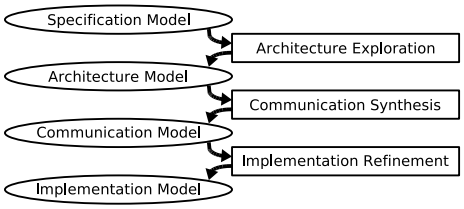


Fig. 1. Design Process for Embedded Systems

and time-triggered communication. This paper also includes system level interpretation of the communication models using the SpecC language so as to verify them. SpecC is used because it is tightly coupled with the above-mentioned design process and methodology. Codes of the models could be easily translated into SystemC or SystemVerilog.

Section 2 summarizes event-triggered and time-triggered communication. Section 3 proposes stepwise exploration of communication. Section 4 and 5 verifies the communication models in SpecC. Section 6 mentions some related works, and contains concluding remarks.

2 Event-Triggered and Time-Triggered Communication

There are two major categories for network communication in distributed embedded systems: event-triggered and time-triggered. Event-triggered communication is flexible, and appropriate for soft real-time systems. Time-triggered communication, on the contrary, is deterministic, in the sense that all instants of message transmission are scheduled beforehand. This is suitable for applications in which the data traffic is of a periodic nature, and this ensures dependable hard real-time message delivery which is necessary in safety-critical applications.

Both the above have two sub-categories: centralized and decentralized. In centralized event-triggered/time-triggered communication, a single arbiter/scheduler manages the whole network. In decentralized event-triggered/time-triggered communication, each module is responsible for arbitration/scheduling. The former is less expensive and easier to implement, while the latter is faster, and more robust due to the absence of a single point prone to failures and load concentration.

For example, below are some protocols for in-vehicle networks:

- CAN (Controller Area Network) [4,5,6]: a broadcast, differential serial bus standard. Its bit rates is up to 1 Mbps. It is decentralized event-triggered.
- LIN (Local Interconnect Network) [7]: also a broadcast serial network. It is designed as a small and economical substitute for CAN, and its bit rates is up to 20 kbps. It is centralized time-triggered.
- FlexRay [8]: next generation automotive network communications protocol. its bit rates is up to 20 Mbps. It is decentralized time-triggered.

3 Stepwise Exploration of Communication

In the conventional design process, we must select which communication protocol to use at the beginning. Once having selected any, we cannot switch to another, even if it is found later that another is better. A complex distributed embedded system may include several categories of communication, which should be selected depending on physical constraints, thus it is sometimes difficult or unable to select at the beginning. In addition, we cannot reuse a component or framework for other systems based on any other protocols.

Consequently, referring Model Driven Architecture (MDA) discipline [9], which is now widely accepted in Software Engineering, we investigate a design process of communication which begins from abstract specification without assuming any communication category, then explores the categories in a stepwise manner, and is followed by physical implementation synthesis. Fig. 2 shows the result in outline.

At the beginning, sender and receiver modules are connected by an abstract channel with virtual functions of sending and receiving. Then, the channel is transformed into a more concrete model, and there are two choices: an event-triggered channel or a time-triggered channel. The modules need no transformation. The event-triggered channel is accompanied with virtual arbiter and filter, while the time-triggered channel is with a virtual scheduler. Next come a centralized or decentralized model for each event-triggered and time-triggered channel. In the decentralized model, the function of arbitration, filtering, and scheduling are embedded in the sender and receiver modules.

At each model, a designer verifies its correctness, and then selects which way to go, considering advantages of each path such as presented in Section 2, based on some estimation or profiling which reflect requirements and constraints. The designer makes decision, not at once at the beginning, but in a stepwise manner gradually fixing details. Also, the designer verifies the system, not at once after implements it, but in a stepwise manner.

4 SpecC Interpretation of Communication Models

This section presents interpretation of each of the seven communication models mentioned above. The results are codes in SpecC, which can be executed and verified. Here, only essential parts of some codes are shown.

(1) Abstract Communication Model: Each pair of sender-receiver has its own virtual communication line which is simulated by a shared variable with synchronization (Fig. 3). `I_snd` and `I_rcv` are interfaces of `Chnl` which connects two behaviors `Sender` and `Receiver`. `Chnl` has an array of shared variables `line`, each of which simulates a communication like corresponding to each `Receiver` instance. `ID`'s are assigned to the behaviors elsewhere. An array of event `e` is for synchronization between the `Sender` and `Receiver`.

(2) Event-Triggered Communication Model: The virtual and per-ID synchronization is replaced by an arbiter and a filter for event-triggered

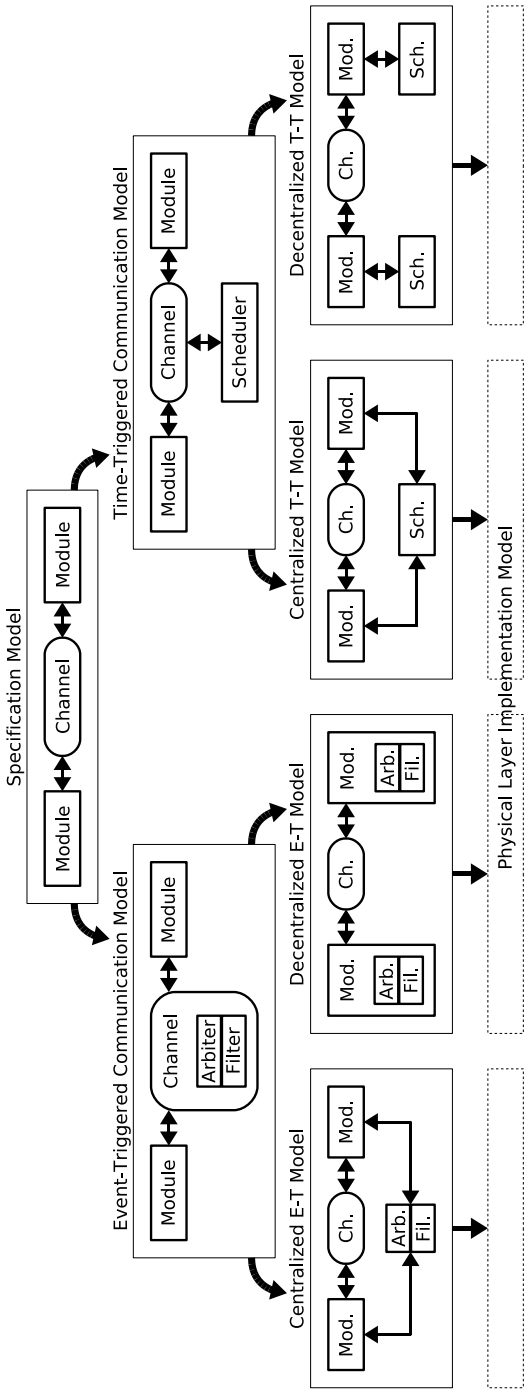


Fig. 2. Stepwise Exploration of Communication Models

```
interface I_snd {
  void send(ID i,DATA d); };

interface I_rcv {
  DATA receive(ID i); };

channel Chnl(void)
  implements I_snd,I_rcv {
  DATA line[MAX];
  event e[MAX];

  void send(ID i,DATA d) {
    line[i]=d;
    notify(e[i]); }

  DATA receive(ID i) {
    wait(e[i]);
    return line[i]; } };

behavior Sender(I_snd s) {
  ID receiver_id;
  DATA d;
  void main(void) {
    ...
    s.send(receiver_id,d); } };

behavior Receiver(I_rcv r) {
  ID my_id;
  DATA d;

  void main(void) {
    d=r.receive(my_id);
    ... } };

behavior System(void) {
  Chnl ch;
  Sender s1(ch);
  Receiver r1(ch);

  void main(void) {
    par{
      s1.main();
      r1.main(); } } };
```

Fig. 3. SpecC Code for Abstract Communication Model

communication (Fig. 4). The sender's and receiver's behaviors remain the same as in (1), and are omitted from the figure. Now, the `Chnl` contains an `Arbiter` and `Filter` so that all the communications share a single line. The variable `use` is set when any communication occupies the line.

(3) Time-Triggered Communication Model: The virtual synchronization is replaced by a scheduler for time-triggered communication (Fig. 5). The sender's and receiver's behaviors remain the same as in (1), and are omitted from the figure. Now, the `Chnl` shares with the `Scheduler` a variable `slot_id`, which indicates a scheduled time slot assigned to an `ID` for communication.

(4) Centralized Event-Triggered Communication Model: The arbiter and the filter are raised from the inner behavior within the channel to the top-most behavior. The structure of codes remain almost the same as in (2), and is omitted in this paper.

(5) Decentralized Event-Triggered Communication Model: The arbiter and filter are duplicated, and placed into the sender's and receiver's behaviors respectively (Fig. 6). The channel is replaced by a simple wire for transmission, and the senders and the receivers share this single wire. The receiver `senses` the wire to identify whether the communication is to itself or not.

(6) Centralized Time-Triggered Communication Model: The scheduler is coordinated with the sender and receiver, not with the channel as in (3). The structure of codes remain almost the same, and are omitted in this paper.

```
struct PACKET {DATA d,ID i};  channel Chnl(void)
                                implements I_snd,I_rcv {
channel Arbiter(void)           Arbiter arb;
  implements I_set,I_reset {    Filter fil;
  int use=0;                    struct PACKET p;

  void set(void) {              void send(ID i,DATA d) {
    while (use) {                 arb.set();
      waitfor(1); }               p.d=d;
    use=1; }                      p.i=i; }
  void reset(void) {            DATA receive(ID i) {
    use=0; } };                   while (!fil.check(i,p.i)) {
                                    waitfor(1); };
channel Filter(void)              arb.reset();
  implements I_check {            return p.d; } };

  int check(ID i1,ID i2) {
    return(i1==i2); } };
```

Fig. 4. SpecC Code for Event-Triggered Communication Model

```
behavior Scheduler(                 channel Chnl(in ID slot_id)
    out ID slot_id) {                 implements I_snd,I_rcv {
  ID table[MAX];                      DATA slot;
  int k;
                                      void send(ID i,DATA d) {
  void main(void) {                     while (i!=slot_id) {
    k=0; while (1) {                      waitfor(1); };
      slot_id=table[k];                 slot=d; }
      k++;                            DATA receive(ID i) {
      if (k>=MAX) { k=0; };             while (i!=slot_id) {
      waitfor(1); } } };                  waitfor(1); };
                                        return slot; } };
```

Fig. 5. SpecC Code for Time-Triggered Communication Model

```
behavior Sender(I_wire w) {         behavior Receiver(I_wire w) {
  ID receiver_id;                     ID my_id,i;
  DATA d;                             DATA d;

  void main(void) {                   void main(void) {
    ...                                 while ((i=w.sense())
    while (w.sense()) {                     && (my_id != i)) {
      waitfor(1); }                       waitfor(1); };
    w.transmit(receiver_id);            d=w.sense()
    w.transmit(d); } };                 ... } };
```

Fig. 6. SpecC Code for Decentralized Event-Triggered Communication Model

```
behavior Sender(I_wire w) {
  Scheduler sch(slot_id);
  ID receiver_id;
  DATA d;

  void main(void) {
    par{
      sch.main();
    {
    ...
    while (receiver_id
         !=slot_id) {
      waitfor(1); };
    w.transmit(d); } } } };
```

```
behavior Receiver(I_wire w) {
  Scheduler sch(slot_id);
  ID my_id;
  DATA d;

  void main(void) {
    par{
      sch.main();
    {
    while (my_id
         !=slot_id) {
      waitfor(1); };
    d=w.sense();
    ... } } } };
```

Fig. 7. SpecC Code for Decentralized Time-Triggered Communication Model

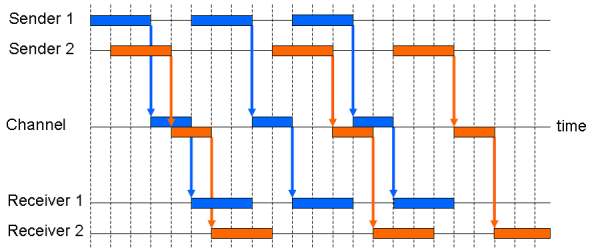
(7) Decentralized Time-Triggered Communication Model: The scheduler is duplicated, and moved into the sender's and receiver's behaviors (Fig. 7). The channel is replaced by a simple wire for transmission, and the senders and the receivers share this single wire. The sender's `Scheduler` instance and the receiver's `Scheduler` instance synchronizes so that they give consistent scheduling.

```
abstract model:
0000 Sender1: start
0001 Sender2: start
0003 Sender1: send
0003 Channel: Sender1 to Receiver1 start
0004 Sender2: send
0004 Channel: Sender2 to Receiver2 start
0005 Sender1: start
0005 Channel: Sender1 to Receiver1 end
0005 Receiver1: receive
0006 Channel: Sender2 to Receiver2 end
0006 Receiver2: receive
0008 Sender1: send
0008 Channel: Sender1 to Receiver1 start
0009 Sender2: start
0010 Sender1: start
0010 Channel: Sender1 to Receiver1 end
0010 Receiver1: receive
0012 Sender2: send
0012 Channel: Sender2 to Receiver2 start
0013 Sender1: send
0013 Channel: Sender1 to Receiver1 start
0014 Channel: Sender2 to Receiver2 end
0014 Receiver2: receive
```

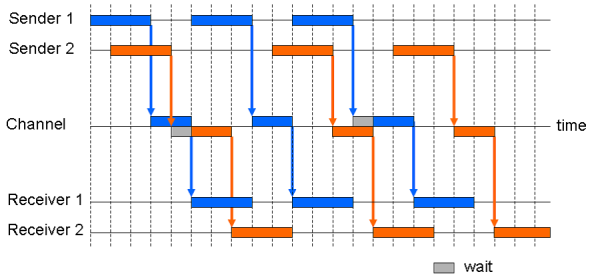
Fig. 8. Execution Log of the Abstract Model

5 Experiments

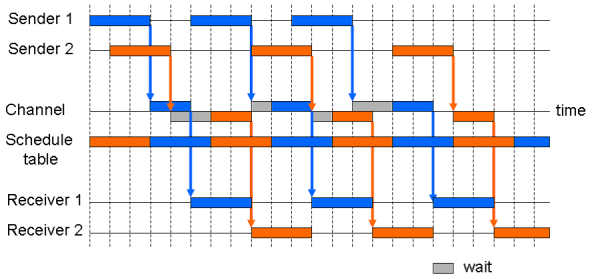
We have verified all the seven models appeared above, by implementing all the full codes. All the models must preserve the same behavior (in the general sense) as the abstract communication model that sends and receives data in an arbitrary order. The event-triggered communication model and its centralized and decentralized sub-models use a single wire which all the sender-receiver pair share by arbitration as well as preserving the behavior of the abstract communication model. The time-triggered communication model and its centralized and decentralized sub-models use a single wire, not by arbitration but by time-slicing scheduling.



(a) Abstract Model



(b) Event-Triggered Model



(c) Time-Triggered Model

Fig. 9. Execution Sequences of Design Process for Embedded Systems

Here we present some execution logs. Fig. 8 is an log extract of the abstract model. Fig 9 is a set of schematic sequence representations of logs of the abstract model, the event-triggered model, and the time-triggered model, respectively. These confirm that our models work properly.

6 Concluding Remarks

Stepwise exploration encourages stepwise decision making, component and framework reuse, and early stage verification, all of which accelerate design processes. This paper applies it to design of distributed embedded systems, as the first step to communication exploration. This paper also contributes toward integrated design of event-triggered and time-triggered communication, which are used separately at present.

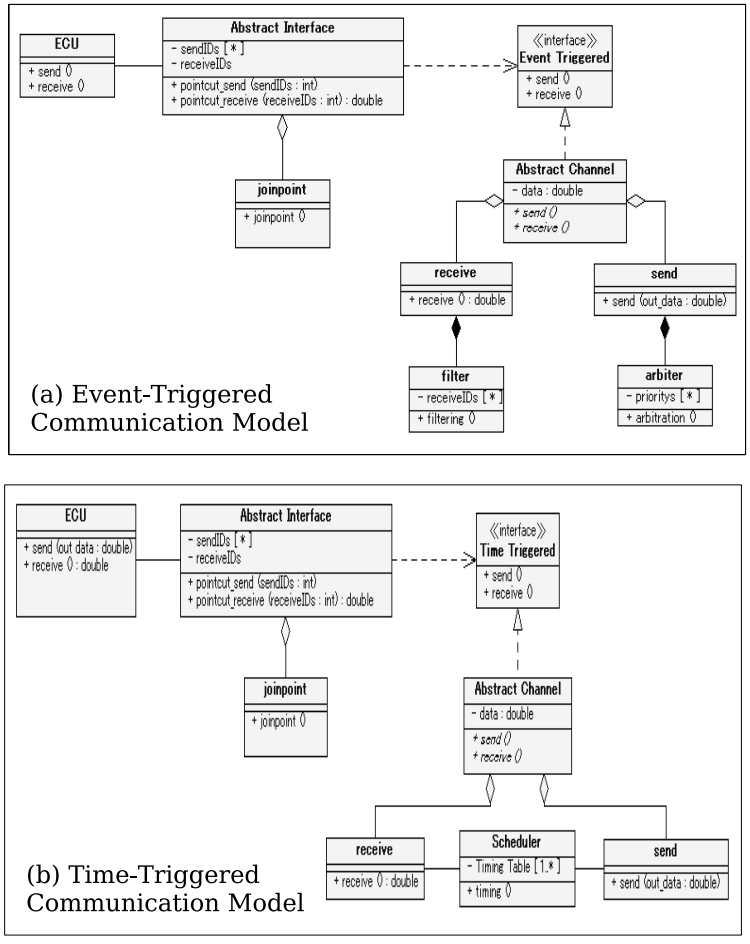


Fig. 10. UML Class Diagrams of Communication Models

There have been some researches on mixed scheduling of event-triggered and time-triggered communication [10,11], however there has been none yet on communication exploration in distributed embedded system design.

Our ongoing studies are interpreting them in Executable UML [9]. There is a research trend to apply UML and MDA to System-Level Design from a few years ago [12]. However, there has been no study on communication exploration yet.

Some models described in UML are shown in Fig. 10 as examples. The former is the class diagram for event-triggered communication model, and the latter is the one for time-triggered. "EUC" (Electronic Control Unit) is an embedded module, and "joinpoint" is a tool class for framework reuse.

We are still at the starting point of this study. Our ongoing studies are: (1) interpreting them in Executable UML as mentioned above, (2) formalizing semantics-preserving transformation between models to build an automatic CAD tool, and (3) investigating some real-world applications.

References

1. Gajski, D.D., et al.: SpecC: Specification Language and Methodology. Kluwer, Dordrecht (2000)
2. Gerstlauer, A., et al.: System Design: A Practical Guide with SpecC. Kluwer, Dordrecht (2001)
3. Kobayashi, K., et al.: Exploration of Communication Models in the Design of Distributed Embedded Systems. IEEJ Trans. on Electrical and Electronic Engineering 2(3), 402–404 (2007)
4. Robert Bosch GmbH, CAN Specification (1991)
5. ISO TC 22/SC 3, Controller Area Network (CAN), ISO 11898 (2003)
6. ISO TC 22/SC 3, Low-Speed Serial Data Communication, ISO 11519 (2005)
7. LIN Consortium, LIN Specification (1999)
8. FlexRay Consortium, FlexRay Protocol Specification (2005)
9. Mellor, S.J., et al.: MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley, Reading (2004)
10. Pop, T., et al.: Schedulability Analysis for Distributed Heterogeneous Time/Event Triggered Real-Time Systems. In: Proc. IEEE 15th Euromicro Conference on Real-Time Systems, pp. 257–266 (2003)
11. Pop, P., et al.: Schedulability-Driven Partitioning and Mapping for Multi-Cluster Real-Time Systems. In: Proc. IEEE 16th Euromicro Conference on Real-Time Systems, pp. 91–100 (2004)
12. Proc. 2006. Workshop on UML for SoC Design, in conjunction with ACM/IEEE 43th Design Automation Conf. (2006)