Specifications, Modeling, and Model of Computation

Jian-Jia Chen
(slides are based on Peter Marwedel)
TU Dortmund, Informatik 12

2017年10月11日
Hypothetical design flow

Numbers denote sequence of chapters

2: Specification
3: ES-hardware
4: System software (RTOS, middleware, …)
5: Evaluation & Validation (energy, cost, performance, …)
6: Application mapping
7: Optimization
8. Test *

* Could be integrated into loop
Motivation for considering specs & models

- Why considering specs and models in detail?

- If something is wrong with the specs, then it will be difficult to get the design right, potentially wasting a lot of time.

- Typically, we work with models of the system under design (SUD)

- What is a model?
Models

Definition: A model is a simplification of another entity, which can be a physical thing or another model. The model contains exactly those characteristics and properties of the modeled entity that are relevant for a given task. A model is minimal with respect to a task if it does not contain any other characteristics than those relevant for the task.

[Jantsch, 2004]:

Which requirements do we have for our models?
Requirements for specification & modeling techniques: Hierarchy

Hierarchy
Humans not capable to understand systems containing more than ~5 objects. Most actual systems require more objects

Hierarchy (+ abstraction)

- Behavioral hierarchy
  Examples: states, processes, procedures.

- Structural hierarchy
  Examples: processors, racks, printed circuit boards
- Systems must be designed from components
- Must be “easy” to derive behavior from behavior of subsystems
- Work of Sifakis, Thiele, Lee, Lee, Ernst, …
- Concurrency
- Synchronization and communication
Requirements for specification & modeling techniques (3): Timing

- **Timing behavior**
  Essential for embedded and cy-phy systems!
  - Additional information (periods, dependences, scenarios, use cases) welcome
  - Also, the structure of the underlying platform must be known
Requirements for specification & modeling techniques (3): Timing (2)

4 types of timing specs required, according to Burns, 1990:

1. Measure elapsed time
   Check, how much time has elapsed since last call

   ![Diagram showing elapsed time]

2. Means for delaying processes

   ![Diagram showing delaying processes]
3. **Possibility to specify timeouts**
   Stay in a certain state a maximum time.

4. **Methods for specifying deadlines**
   Not available or in separate control file.
Specification of ES (4): Support for designing reactive systems

- **State-oriented behavior**
  Required for reactive systems; classical automata insufficient.

- **Event-handling**
  (external or internal events)

- **Exception-oriented behavior**
  Not acceptable to describe exceptions for every state
Requirements for specification & modeling techniques (5): More Features

- Presence of programming elements
- **Executability** (no algebraic specification)
- Support for the design of large systems (OO)
- Domain-specific support
- Readability
- Portability and flexibility
- Termination
- Support for non-standard I/O devices
- Non-functional properties
- Support for the design of dependable systems
- No obstacles for efficient implementation
- Adequate model of computation
Then, Always Remember

Concrete System

Concrete problem instance
messy infeasible
Solution to concrete problem

Concretization

Models

Abstract problem instance
elegant insightful
Solution to abstract problem

abstraction and Model

Concretized abstract solution
Problems with classical CS theory and von Neumann (thread) computing

Even the core … notion of “computable” is at odds with the requirements of embedded software. In this notion, useful computation terminates, but termination is undecidable.

In embedded software, termination is failure, and yet to get predictable timing, subcomputations must decidably terminate.

What is needed is nearly a reinvention of computer science.


Search for non-thread-based, non-von-Neumann MoCs.
Models of computation

What does it mean, “to compute”? Models of computation define:

- Components and an execution model for computations for each component
- Communication model for exchange of information between components.
Dependence graph: Definition

**Def.: A dependence graph** is a directed graph $G=(V,E)$ in which $E \subseteq V \times V$ is a relation.

If $(v_1, v_2) \in E$, then $v_1$ is called an **immediate predecessor** of $v_2$ and $v_2$ is called an **immediate successor** of $v_1$.

Suppose $E^*$ is the transitive closure of $E$.

If $(v_1, v_2) \in E^*$, then $v_1$ is called a **predecessor** of $v_2$ and $v_2$ is called a **successor** of $v_1$. 

Nodes could be programs or simple operations.
Dependence graph: Timing information

Dependence graphs may contain additional information, for example:

- Timing information

![Dependence graph]

Arrival time: (0,7]  deadline: (1,8]  (3,10]
Dependence graph: I/O-information
Dependence graph: Hierarchical task graphs
Dependence graph: Shared resources
Dependence graph: Periodic schedules

- A job is single execution of the dependence graph
- Periodic dependence graphs are infinite
Communication

- **Shared memory**

![Diagram showing shared memory between Comp-1, memory, and Comp-2]

Variables accessible to several components/tasks.

Model mostly restricted to local systems.
Shared memory

```
thread a {
    u = 1; ..
P(S) //obtain mutex
if u<5 {u = u + 1; ..}
// critical section
V(S) //release mutex
}

thread b {
    ..
P(S) //obtain mutex
u = 5
// critical section
V(S) //release mutex
}
```

- Unexpected u=6 possible if P(S) and V(S) is not used (double context switch before execution of \{u = u+1\})
- S: semaphore
- P(S) grants up to \(n\) concurrent accesses to resource
- \(n=1\) in this case (mutex/lock)
- V(S) increases number of allowed accesses to resource
- Thread-based (imperative) model should be supported by mutual exclusion for critical sections
Exercise

\begin{verbatim}
thread a {
    u = 3;                 //op a1
    P(s);
    if u<10               //op a2
        {u = u + 1; ..}  //op a3
    else u=5;          //op a4
    V(s);
}

thread b {
    u = 2;                //op b1
    P(s);
    if u<4                 //op b2
        {u = u + 4; ..}  //op b3
    else u=10;        //op b4
    V(s);
}
\end{verbatim}

• Each thread is supposed to be executed once.
• Threads can be executed in any sequence and the execution can switch between the two threads at any time.
• For the sake of simplicity, we assume that each operation is executed in an atomic manner, i.e. there is no switch (called context switch) during the execution of ax and thread by.
• Which values of u are possible at the completion of both tasks?
Non-blocking/asynchronous message passing

Sender does not have to wait until message has arrived;

Potential problem: buffer overflow
Blocking/synchronous message passing - *rendez-vous*

Sender will wait until receiver has received message

```
... send() ...
```

```
... receive () ...
```

No buffer overflow, but reduced performance.
Organization of computations within the components (1)

- Finite state machines
Organization of computations within the components (2)

- Discrete event model

<table>
<thead>
<tr>
<th>a</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>7</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
</tr>
</tbody>
</table>

queue

<table>
<thead>
<tr>
<th>a:=5</th>
<th>b:=7</th>
<th>c:=8</th>
<th>a:=6</th>
<th>a:=9</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
<td>13</td>
<td>15</td>
<td>19</td>
</tr>
</tbody>
</table>

- Von Neumann model

Sequential execution, program memory etc.
Organization of computations within the components (3)

- Differential equations

\[ \frac{\partial^2 x}{\partial t^2} = b \]

- Data flow
  (models the flow of data in a distributed system)

- Petri nets
  (models synchronization in a distributed system)
Summary

Requirements for specification & modeling

- Hierarchy
- ..
- Appropriate model of computation

Models of computation =

- Dependence graphs
- models for communication
- models of components