Embedded Systems: Specification and Modeling (part I)

Todor Stefanov
Leiden Embedded Research Center,
Leiden Institute of Advanced Computer Science
Leiden University, The Netherlands
Outline

Why considering modeling and specification?

Requirements for Specification Techniques

Models of Computation
- State-based models (not considered in this course!)
  - FSM (classical automata)
  - Timed automata
  - StateCharts
- Petri Nets (not considered in this course!)
  - Condition/Event Nets
  - Predicate/Transition Nets
  - Place/Transition Nets
- Actor-based Dataflow models
  - SDF, CSDF, PPN, PSDF, PCSDF, PPPN, BDF, DDF, KPN

Specification Languages
- VHDL, SystemC, SpecC, Others
Why considering specifications?

- The first step in designing Embedded System is to precisely tell *what the system behavior should be*.

**Specification:** correct, clear and unambiguous description of the required system behavior.

- This can be extremely difficult
  - Increasing complexity of Embedded Systems
  - Desired behavior often not fully understood in the beginning
- If something is wrong with the specification, then it will be difficult to get the design right, potentially wasting a lot of time.
- How can we (correctly and precisely) capture systems behavior?
Model-based Specifications

- Typically, we work with models of the system under design at different levels of abstraction.
- Levels of abstraction alleviate the complexity problem of specification.
  - Levels of abstraction will be discussed later.
- Models allow to reason about the systems under design, thereby identifying and correcting flaws in the specification.
- What is a model anyway?
**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]

- What are the requirements for Model-based Specification techniques for Embedded Systems?
Requirements for Model-based Specification Techniques (1)

- Modularity
  - Systems specified as a composition of objects
  - Hierarchical composition of objects
    - Most humans not capable to understand systems containing more than ~5 objects BUT most actual systems require more objects
      - Example1: statements -> procedures -> programs
      - Example2: transistors -> gates -> functional blocks
  - It must be “easy” to derive systems behavior from behavior of subsystems

- Concurrency, synchronization and communication
Requirements for Model-based Specification Techniques (2)

- **Timing behavior**
  - Essential for embedded systems!!!
  - Four types of timing specs required, according to Burns, 1990:

1. **Techniques to measure elapsed time**
   - Check, how much time has elapsed since some computation has happened

2. **Means for delaying processes**

3. **Possibility to specify timeouts**
   - Stay in a certain state a maximum time

4. **Methods for specifying deadlines**
Requirements for Model-based Specification Techniques (3)

- Support for designing reactive systems, i.e., means to describe:
  - State-oriented behavior
  - Event-handling (external or internal events)
  - Exception-oriented behavior
- Readability, Portability, and Flexibility
- Termination
  - It should be clear, at which time all computations are completed
- Non-functional properties
  - Power consumption, area, cost, etc.
- No obstacles for efficient implementation
- Simplicity
Models of Computation

- Models of Computation (MoC) define:
  - Components and an execution model for computations for each component
  - Communication model for exchange of information between components

There is **NO** model of computation that meets all specification requirements previously discussed.

- Thus, selecting appropriate MoCs for specifying a system is the key to successful and efficient design of Embedded Systems
Is Van Neumann MoC appropriate?

- An instruction set, a memory, and a program counter, is all we need to execute whatever application we can dream of.
- Von Neumann model of computation does not match well with requirements for embedded system design.
- This model does not consider timing requirements and constraints, e.g.,
  - Timing cannot be described (instructions cannot be delayed or forced to execute at a specific time).
  - Timing deadlines cannot be specified for instructions or sequence of instructions.
  - Timeouts cannot be specified for sequence of instructions.
Another inappropriate MoCs: Thread-based concurrency models

“… threads as a concurrency model are a poor match for embedded systems. … they work well only … where best-effort scheduling policies are sufficient.”


- Thread-based multiprocessing may access global variables

- We know from the theory of operating systems that
  - Access to global variables might lead to race conditions
  - To avoid these, we need to use mutual exclusion
  - Mutual exclusion may lead to deadlocks
Problems with thread-based concurrency

- Threads are wildly nondeterministic
- The programmer’s job is to prune away the nondeterminism by imposing constraints on execution order (e.g., mutexes)

  Nontrivial software written with threads, semaphores, and mutexes is incomprehensible to humans

- Thus,
The bottom line is

When specifying and designing Embedded Systems we should search for and use NON-thread-based, NON-von-Neumann Models of Computation.

- Finding an appropriate model to capture embedded system’s behavior is an important step
  - Model shapes the way we think of the system
  - For control-dominated and reactive systems
    - State-based models are appropriate, monitor control inputs and set control outputs
  - For data-dominated systems
    - Actor-based dataflow models are appropriate, transform input data streams to output data streams
Actor-based Models of Computation: Terminology

- **Actor-based MoC**
  - Formal representation of operational semantics of a network of functional blocks describing the computation

- **Actor**
  - Describes some functionality

- **Relation**
  - The actors can communicate with each other using relations

- **Token**
  - A quantum of information that is exchanged

- **Firing of actor**
  - A quantum of computation
  - Moment of interaction with other actors

```plaintext
fire {
  ...  
  token = get();
  ...  
  send(token);
  ...  
}
```

**Diagram**

- Network
- Actor
- Port
- Token
Active/Passive Actors

- **Passive Actor:**
  - Scheduler needed to activate the firing
  - Schedule ABBCD
  - A firing needs to terminate
  - Fire-and-exit behavior

- **Active Actor:**
  - Schedules itself
  - A firing typically does not terminate
    - Endless while loop
  - Process behavior

Two kinds of Actors:

```java
fire {
  token = get();
  ...
  send(token);
  ...
}
```

```java
fire {
  while(1) {
    token = get();
    send(token);
  }
}
```
Communication Between Actors

- Data Type of the Token
  - Integer, Double
  - Complex
  - Matrix, Vector
  - Record

- Way exchange takes place
  - Buffered
  - Timed
  - Synchronized

Communication (Semantics)
Different (Communication) Semantics

- The way the exchange of the tokens takes place, distinguishes a MoC
  - Discrete-event models (DE)
    - Global notion of time
    - Actors are fired at time instances by a global scheduler
    - Tokens carry time-tags that express when an actor needs to be activated
  - Synchronous-reactive models (SR)
    - Co-design finite state machines (CFSM)
      - Presence of a token represents that an event took place
      - Actor has to react on the event
  - Dataflow models
    - Tokens represent the fulfillment of data dependencies
    - Tokens exchanged between actors are buffered
Dataflow Models (1)

- Network of concurrently executing processes/actors
  - Dataflow Processes/Actors
    - Passive or Active
    - Can be described with imperative code
  - Dataflow Communication
    - Only through FIFO buffers
    - Buffers usually treated as unbounded for flexibility
    - Sequence of tokens read guaranteed to be the same as the sequence of tokens written
    - Destructive read: reading a token from a buffer removes the token
    - Much more predictable than shared memory

```plaintext
fire {
  ... send();
  ... send();
}

port

fire {
  ... get();
  ... send();
}

port

fire {
  ... get();
  ... get();
}
```

Process 1  

FIFO  

Process 2  

FIFO  

Process 3
Dataflow Models (2)

- Data-driven execution model
  - A process/actor can fire whenever it has sufficient data on its input FIFO buffers
  - The order in which processes/actors execute is not part of the model specification
  - The order is typically determined by a compiler (at design time), the hardware or OS (at run-time), or both

- Applications of Dataflow models
  - Anything that deals with a continuous stream of data
  - Well suited to model streaming data applications (signal processing, multimedia, etc.)
  - Perfect fit for graphical (block-diagram) specifications
Dataflow Features and Advantages

- Exposes coarse-grain parallelism
- Exposes high-level structure that facilitates analysis, verification, and optimization
- Complementary to ongoing advances in DSP compiler technology for procedural languages, such as C and MATLAB
- Encourages desirable software engineering practices: modularity and code reuse
- Intuitive to signal processing and multimedia algorithm designers
- Widely used in academic and industrial design tools
Examples of tools using Dataflow Models of Computation

- **Agilent ADS tool**

- **Ptolemy II** (UC Berkeley) is an environment for simulating multiple models of computation
  - http://ptolemy.berkeley.edu/

![Diagram of Ptolemy II](image)
Dataflow Modeling Space

- **Expressiveness:**
  - Indicate what type of systems can be modeled and how compact the model is

- **Analyzability:**
  - The degree of possibility for compile-time analysis (scheduling, buffer sizes, etc.)

- **Implementation efficiency:**
  - Influenced by the required scheduling policy, code size, and the need of run-time support

**Decidable Models:**
- Synchronous Data Flow (SDF)
- Homogeneous SDF (HSDF)
- Multi-Dimensional SDF (MDSDF)
- Cyclo-Static Data Flow (CSDF)
- Polyhedral Process Network (PPN)

**Partly-Decidable Models:**
- Parameterized [SDF, CSDF, PPN]

**Undecidable Models:**
- [Boolean, Dynamic] Data Flow
- [Dataflow, Kahn] Process Network

(*Third dimension could be added: simplicity and intuitive appeal*)
Decidable Dataflow Models

- Used for representing static data flow behavior:
  - Polyhedral Process Networks (PPN), compact multiphase modeling →
  - Cyclo-static dataflow (CSDF), multiphase modeling →
  - Synchronous dataflow (SDF), multirate modeling →
  - Homogeneous synchronous dataflow (HSDF), single rate modeling

- These are in decreasing order of generality

- Designs represented in more general models can be converted to equivalent representations in less general ones BUT there is a complexity explosion due to the conversion
  - e.g., PPN → CSDF → SDF → HSDF

- Compile-time techniques
  - Static scheduling: low overhead, predictability
  - Performance analysis
  - Loop scheduling techniques for
    - Performance speedup
    - Code/data minimization
    - Hierarchical parallel scheduling
  - Task scheduling for latency/throughput optimization
Synchronous Data Flow (SDF)

- Introduced by Lee and Messerchmitt, UC Berkeley, 1987
- Network of concurrent executing actors
  - Passive actors
  - Communication is buffered
- The model progresses as a sequence of “iterations”.
- A “firing rule” determines the firing condition of an actor.
- At each firing, a fixed number of tokens is consumed and produced.
- Characteristics of SDF
  - Compile time analyzable.
  - Static schedule
  - Optimization for memory/speed
  - Static dataflow

Iteration: ABBBCD
SDF Operational Semantics: Firing Rule

- An actor of SDF is \textit{enabled} if there is a certain number of tokens on each of its input arcs.
- An \textit{enabled actor is fired} by removing a number of tokens from each of its input arcs and placing tokens on each of its output arcs.
- Iteration: a sequence of actors’ firings that brings the SDF network to its initial state.
  - Many possible sequences as long as firing rules are obeyed.

Iteration: ABBBCD

![Diagram of SDF network with tokens and connections]

Iteration: ABBBCD
SDF: Fixed Production and Consumption Rate

Balance equations (one for each channel):

\[ f_A N = f_B M \]

- Schedulable statically
- Decidable:
  - buffer memory requirements
  - deadlock

number of tokens consumed
number of firings per “iteration”
number of tokens produced
SDF: Applications

- Restriction: Fixed production and consumption rate
  - BUT natural for modeling multi-rate signal processing

- Typical signal-processing blocks are:
  - Unit-rate
    - Adders, multipliers
  - Upsamplers (1 in, n out)
  - Downsamplers (n in, 1 out)

- Example: DAT-to-CD rate converter
  - Converts a 44.1 kHz sampling rate to 48 kHz

\[
(44.1 \times 2/1 \times 2/3 \times 8/7 \times 5/7 = 48)
\]
SDF: Initial Tokens

- SDF model does not have an initialization phase
- Alternative: an SDF model may start with tokens in its buffers
- These behave like delays (signal-processing)
- Delays are sometimes necessary to avoid deadlock
- Example: Finite Impulse Response (FIR) Filter

```
dup *c dup *c dup *c dup *c +
```

One-cycle delay

**Duplicate**

**Constant multiply (filter coefficient)**

**Adder**
SDF: Scheduling

- Schedule can be determined completely at compile-time, i.e., before the system runs
  - Two steps:
    1. Establish relative firing rates of actors by solving a system of linear equations
    2. Determine periodic schedule by simulating system for a single iteration

- Goal: a sequence of actor firings that
  - Runs each actor at least once in proportion to its rate
  - Avoids underflow
    - no actor fired unless all tokens it consumes are available
  - Returns the number of tokens in each buffer to their initial state

- Result: the schedule can be executed repeatedly without accumulating tokens in buffers
Step 1: Calculating Rates (1)

- Each arc imposes a constraint
  - The number of tokens produced should be equal to the number of the tokens consumed
  - The balance equation guarantees this for each arc
- Example:

  \[
  \begin{align*}
  3a - 2b &= 0 \quad \text{(for arc ab)} \\
  4b - 3d &= 0 \quad \text{(for arc bd)} \\
  b - 3c &= 0 \quad \text{(for arc bc)} \\
  2c - a &= 0 \quad \text{(for arc ca)} \\
  d - 2a &= 0 \quad \text{(for arc da)}
  \end{align*}
  \]

  Solution:
  
  \[
  \begin{align*}
  a &= 2c \quad \text{(a should fire twice more than c)} \\
  b &= 3c \\
  d &= 4c
  \end{align*}
  \]
Step 1: Calculating Rates (2)

- **Consistent systems:**
  - Have more than one solution (all-zeros solution + other solutions)
  - Usually we want the smallest integer non-all-zeros solution

- **Inconsistent systems:**
  - Have only the all-zeros solution

- **Disconnected systems:**
  - Relative rates between some actors undefined

- **Example: Consistent Systems**

\[
\begin{align*}
3a - 2b &= 0 \quad \text{(for arc ab)} \\
4b - 3d &= 0 \quad \text{(for arc bd)} \\
b - 3c &= 0 \quad \text{(for arc bc)} \\
2c - a &= 0 \quad \text{(for arc ca)} \\
d - 2a &= 0 \quad \text{(for arc da)}
\end{align*}
\]

This is the smallest integer solution which is non-zero

\[
a = 2 \quad b = 3 \quad c = 1 \quad d = 4
\]

**Solution:**

\[
\begin{align*}
a &= 2c \quad \text{(a should fire twice more than c)} \\
b &= 3c \\
d &= 4c
\end{align*}
\]
Inconsistent and Disconnected Systems

- Inconsistent system
  - No way to execute it without an unbounded accumulation of tokens on the channels
  - Only solution is “do nothing”, i.e.,
    - The only integer solution is $a=0 \ b=0 \ c=0$

- Disconnected system (under-constrained system)
  - Two or more unconnected pieces
  - Relative rates between pieces undefined

\[
\begin{align*}
  a - c &= 0 \\
  a - 2b &= 0 \\
  3b - c &= 0 \\
  Or \quad 2b - c &= 0 \\
  3b - c &= 0
\end{align*}
\]
Consistent Rates Not Enough

- A consistent system with no schedule
- Rates do not avoid deadlock
- Example: deadlock in consistent system

Solution here: add an initial token (delay) on one of the arcs

Initially No Tokens
a waits for b
b waits for a
Deadlock!!

Initially 1 Token on arc ab
b can fire
Step 2: Fundamental SDF Scheduling Theorem

Theorem guarantees that any valid simulation will produce a schedule

Example:

Rates: a=2  b=3  c=1  d=4

Possible schedules:

BBBCDDDDAA
BDBDBCADDA
BBDDBBDDCAA

… many more

BC … is not valid
SDF: Scheduling Choices

- SDF Scheduling Theorem guarantees a schedule will be found if it exists
- SDF Systems often have many possible schedules
- How can we use this flexibility?
  - Reduced code size
  - Reduced buffer sizes
- Code Generation
  - Often done with prewritten blocks
  - For traditional DSP, handwritten implementation of large functions (e.g., FFT)
  - One copy of each block’s code made for each appearance in the schedule
    - i.e., no function calls
SDF: Code Generation

- Simple approach: the schedule
  BBBCDDDDAA
  would produce code without loops like
  B;
  B;
  B;
  C;
  D;
  D;
  D;
  D;
  A;
  A;

- Obvious improvement: use loops

- Rewrite the schedule in “looped” form:
  (3 B) C (4 D) (2 A)

- Generated code becomes
  for ( i = 0 ; i < 3; i++) B;
  C;
  for ( i = 0 ; i < 4 ; i++) D;
  for ( i = 0 ; i < 2 ; i++) A;
SDF: Code Size optimization

- Find *Single Appearance Schedule*:
  - (3 B) C (4 D) (2 A)
- Often possible to choose a looped schedule in which each block appears exactly once
- Leads to efficient block-structured code
  - Only requires one copy of each block’s code
- Does not always exist
- Often requires more buffer space than other schedules
- Generated program with efficient code size

```c
for ( i = 0 ; i < 3; i++) B;
  C;
for ( i = 0 ; i < 4 ; i++) D;
for ( i = 0 ; i < 2 ; i++) A;
```
SDF: Buffer Size optimization

- Find *Minimum Memory Schedules*
- Often increases code size (block-generated code)
- Static scheduling makes it possible to exactly predict memory requirements

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Total buffer sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) ABCBCC 3</td>
<td>50 tokens</td>
</tr>
<tr>
<td>(2) A(2B)(4C)</td>
<td>60 tokens</td>
</tr>
<tr>
<td>(3) A(2(B(2C)))</td>
<td>40 tokens</td>
</tr>
<tr>
<td>(4) A(2(BC))(2C)</td>
<td>50 tokens</td>
</tr>
</tbody>
</table>

- Simultaneously improving code size, memory requirements, sharing buffers, etc. remains open research problems
SDF: Parallel Scheduling

SDF is suitable for automated mapping onto parallel processors and synthesis of parallel circuits.

Many scheduling optimization problems can be formulated. Some can be solved, too!
To be continued