Software (System) Synthesis: Raising the Level of Abstraction

Alberto Sangiovanni Vincentelli
Department of EECS
University of California at Berkeley
Qi Zhu
Intel Research, Oregon
Outline

• Lesson learned from logic synthesis
• ‘Software’ Synthesis Flow and Algorithms
• Case Studies
MBD: Code generation

e.g. Mathworks RTW, dSpace TargetLink

High level input models
(Simulink, Modelica, ...)

Direct code generation
- No significant restructuring
- Low level optimization
- Manual partition

Target code
Separation of Concerns (ca. 1990)

Behavior Components
- C-Code
- Matlab
- Dymola

Virtual Architectural Components
- CPUs
- Buses
- Operating Systems

Behavior Platform
- ECU-1
- ECU-2
- ECU-3

Mapping
- Evaluation of Architectural and Partitioning Alternatives

Performance Analysis
- Refinement

Development Process
- Specification
- Analysis
- Implementation
Platform: library of resources defining an abstraction layer with interfaces that allow legal connections

- Resources do contain virtual components i.e., placeholders that will be customized in the implementation phase to meet constraints
- Very important resources are interconnections and communication protocols
Learning from logic synthesis

High level function model

Gate library (platform)

- Separation of func and arch
- Common language for func and arch netlists (Boolean logic, NAND2 gate)
- Automatic mapping

Function model in netlist

Technology Mapping (covering)

Mapped design

Gate library in netlist

Restructuring

Restructuring
Outline

• Lesson learned from logic synthesis
• ‘Software’ Synthesis Flow and Algorithms
• Case Studies
Our software synthesis flow

Stage 1: Common modeling domain (CMD) selection
Common semantics for func and arch
Primitives to decide abstraction level

Stage 2: Automatic mapping

Stage 3: Code generation
Challenges in the flow

• Stage 1: Common modeling domain selection
  ▪ Various models of computation exist in system level.
  ▪ Trade-off between expressiveness and ease of manipulation when selecting the common semantics.
  ▪ Trade-off between granularity and optimality when selecting the primitives.

• Stage 2: Automatic mapping
  ▪ Various constraints and objectives.
  ▪ Domain-specific algorithms may be used albeit not necessary.

• Stage 3: Code generation
  ▪ Communication interface synthesis maybe needed to guarantee correct semantics.
Modeling domain

• **Semantic domain** $Q$ - *the language*
  - Formally defined as trace-based agent algebra [1].
  - $Q.D$: domain of agents - “building blocks”.
  - $Q.A$: master alphabet – “set of all signals between blocks”.
  - $Q.\alpha : Q.D \rightarrow 2^{Q.A}$, each agent has an alphabet – “each block has a set of signals”
  - Operators: renaming, projection and parallel composition – “rules to initialize and compose blocks”

• **Primitives** $P$ – *abstraction level*
  - Primitives are a set of agents in a semantic domain, $P \subseteq Q.D$.
  - No agent in $P$ can be constructed from other agents in $P$.

• **Modeling domain** $C_Q(P)$: all agents constructed from primitives $P$ by applying operators in semantic domain $Q$.

Common modeling domain (CMD)

• A **model** is an agent in the modeling domain.
• Function model $f \in F$, architecture model $a \in A$.
• $B(s)$ denotes the **behavior** of model $s$.
• Modeling domain $M$ is a **common modeling domain** between $f$ and $a$ if there exists $f' \in M$ and $a' \in M$ s.t. $B(f') \subseteq B(f)$ and $B(a') \subseteq B(a)$.
• $f$ and $a$ may have different semantics or abstraction level – hard to explore $o$.
• $f'$ and $a'$ in CMD – mapping space $\Lambda$ can be formally explored.
• $\Lambda \subseteq o$ – mapped behavior is **legal**.
CMD selection

• Ancestor-child relation between modeling domains.
  ▪ Define $\Phi(M) = \{ B(s) \mid s \in C_Q(P) \}$ – set of all agent behavior.
  ▪ $M_1 = C_{Q_1}(P_1)$ is the ancestor of $M_2 = C_{Q_2}(P_2)$ iff $\Phi(M_2) \subseteq \Phi(M_1)$.

• Search CMDs on modeling domain relation graph (directed edges representing ancestor-child relation).

![Diagram with nodes and edges representing CMD selection and modeling domain relations]
CMD selection contd.

- Two design aspects when selecting CMD $C = C_Q(P)$
  - Semantics – decided by semantic domain $Q$
    - Expressiveness vs. analyzability, e.g. dataflow vs. static dataflow.
    - May first choose semantic domain for common ancestor domain $D$, then refine it in $C$.
  - Abstraction level – depends on primitives $P$
    - Explore different abstraction level by choosing different primitives.
    - Carried out when selecting $C$ as child domain of $D$.
  - For both, it is a trade-off between the size of mapping space and complexity.
Covering problem after CMD selection

- Symbols:
  - Function primitive instances: \( F = (f_1, f_2, \ldots, f_n) \)
  - Architecture primitive instances: \( A = (a_1, a_2, \ldots, a_m) \)
  - Mapping decision variables: \( d_{f_i,a_j} \)
  - Architecture selection variables: \( s_{a_j} \)
  - Quantities (power, area, bandwidth...): \( Q_{f_i,a_j,t}, Q_{a_j,t} \)

- General covering formulation

\[ \forall f_i \in F, \quad \sum_{a_j \in A_{f_i}} d_{f_i,a_j} = 1 \]

\[ \forall f_i \in F, a_j \notin A_{f_i}, \quad d_{f_i,a_j} = 0 \]

\[ \forall a_j \in A, \quad \sum_{f_i \in F} d_{f_i,a_j} \geq s_{a_j} \]

\[ \forall f_i \in F, a_j \in A, \quad d_{f_i,a_j} \leq s_{a_j} \]

\[ H_{t,l}(\{d_{f_i,a_j}\}, \{Q_{f_i,a_j,t}\}, \{s_{a_j}\}, \{Q_{a_j,t}\}) \leq 0 \]

\[ \min \ G_t(\{d_{f_i,a_j}\}, \{Q_{f_i,a_j,t}\}, \{s_{a_j}\}, \{Q_{a_j,t}\}) \]

Domain specific. Determines complexity!
Outline

• Lesson learned from logic synthesis
• ‘Software’ Synthesis Flow and Algorithms
• Case Studies
Case study: active safety vehicle

- Functional correctness and cost-efficiency are both important for active safety applications.
- Function and architecture mismatch.

**Function model**
- **synchronous** model.
- no message loss or duplication.

**Architecture platform**
- clock drift between **distributed** ECUs, **asynchronous** communication.
- data loss and duplication exist.

*mismatch*
Stage 1: CMD selection – common semantics

1. Process Networks (PN): expressive but high modeling complexity. Need transformation of both func and arch models.

2. Loosely time triggered architecture (LTTA): transformation of func model to support asynchronous communication. **Chosen in this case study**

3. Synchronous reactive (SR): transformation of the arch to support synchronous communication, by applying following protocols.
   - Clock synchronization.
   - Constraints on task periods.

\[
D = C_{PN}(P_D) \\
F = C_{SR}(P_F) \\
C_1 = C_{LTTA}(P_1 = P_F \cup P_A) \\
A = C_{LTTA}(P_A) \\
C_2 = C_{SR}(P_2 = P_F \cup P_{A_1})
\]
Stage 2: covering problem

Functional Model

- IR Sensor
- Wheel Sensor
- Nav. Task
- Fusion Task
- Object ID Task
- Brake Act.

Signals: tasks, signals

Architectural Model

- ECU1
- BUS1
- ECU2
- BUS2
- ECU3
- ECU4

Varieties: ECUs, messages on buses

Quantity constraints and objective functions
- End-to-end latency
- Utilization
- Extensibility
- ......

Covering variables
- Task to ECU
- Signal to message
- Message selection
- Priority
- Period

Variety of algorithms
- mathematical programming
- heuristics
- meta-heuristics
- machine learning
- ......
Stage 2: covering problem contd.

- Worst case analysis for CAN systems with periodic tasks and messages.
- A complete formulation with all design variables does not scale for industrial size problems.
- We start with tackling following sub-problems.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>Period</td>
<td>Allocation Priority</td>
<td>Allocation Priority</td>
</tr>
<tr>
<td>Objective</td>
<td>Latency</td>
<td>Latency</td>
<td>Extensibility</td>
</tr>
<tr>
<td>Approach</td>
<td>Geometric programming (GP)</td>
<td>Mixed integer linear programming (MILP)</td>
<td>Multi-step Heuristic</td>
</tr>
</tbody>
</table>

Allocation & priority synthesis (MILP based)

**Constraints:**
- End-to-end latency on given paths
- Utilization bound on ECUs and buses

**Objective:**
- Sum of latencies on given paths

**Design inputs:**
- Task worst case execution times
- Task and signal periods
- Architecture topology, bus speeds

**Heuristic:**
- Task and signal priorities

**Step 1:**
Synthesize task allocation (using MILP)

**Step 2:**
Synthesize signal packing, task and message priorities (using MILP)
Allocation & priority synthesis results

Function Model
- 41 Tasks
- 83 Signals
- 171 paths

After mapping:
- Meet all requirements
- Total latency from 36486ms in manual design to 12900ms

Architecture platform
- 9 ECUs
- single bus
Extensibility optimization (MILP and heuristic)

- Initial Task and Signal Priority (Heuristic)
- Initial Task Allocation (MILP)
- Signal Packing and Message Allocation (Greedy Heuristic)
- Task and Message Priority Assignment (Iterative Heuristic)

Flowchart:
- Task Re-allocation (Heuristic for incremental changes)
- Reach Stop Condition?
  - Yes: End
  - No: Repeat process
Extensibility optimization results

• Same active safety vehicle as in allocation and priority synthesis.
• Single-bus and dual-bus options.
• Parameter $K$ to trade off between extensibility and latency.
• Compared with a simulated annealing algorithm: maximum extensibility within 0.3%, runtime 0.5 hour vs. 12 hours.
Case studies in other domains

• Building automation domain [1]
  ▪ Similar semantics as in automotive – synchronous function model and LTTA architecture platform.
  ▪ Also choose SR as the common semantics, however additional timing constraints are added to the architecture for preserving synchronism, as we consider the physical interaction with environment.
  ▪ Mapping leverages COSI for communication network synthesis.

• Multimedia domain [2]
  ▪ JPEG encoder application. Intel MXP architecture platform.
  ▪ Semantics for both function and architecture are dataflow.
  ▪ Challenge is to choose the proper abstraction level. Different levels are explored and compared through choices of primitives.

Concluding remarks

• Software (and hardware) synthesis based on a formal mapping procedure
  ▪ Formally determines the semantics and abstraction level of the design by choosing a common modeling domain.
  ▪ Automatic and optimal mapping algorithms.
  ▪ Generality – applied to various domains with different models of computation as well as different implementation platforms. Domain-specific mapping algorithms may be leveraged in the framework.
  ▪ Optimality – trade-off between complexity and mapping space through the selection of CMD.
  ▪ Reusability – common semantic selection requires designers’ expertise. However proper selection is typically general for particular domains.