Modular code generation from synchronous block diagrams

Stavros Tripakis – Cadence Research Labs
Roberto Lublinerman – Penn State
Context: embedded software

• High-level **modeling** languages, e.g.:
  – Simulink/Stateflow, SCADE, SystemC, …

• Used for modeling/**simulation**, e.g.:
  – Model discrete-time controller + continuous-time plant in Simulink,
  – Simulate and eye-ball to check stability

• But increasingly also for **code generation**:
  – E.g., Real-Time Workshop, dSpace, …
Code generation from synchronous models – previous work

• Synchronous models:
  – Simulink, synchronous languages (Lustre, Esterel, …), …

• Different execution platforms:
  – Single-processor (“centralized”):
    • Single-thread, no RTOS: classic
    • Multi-thread, preemptive scheduling [ACM TECS ’08]
  – Multi-processor (“distributed”):
    • Time Triggered Architecture (TTA) [LCTES’03]
    • Asynchronous networks with bounded FIFO queues [IEEE TC ’08]
    • Loosely TTA [IEEE TC ’08]

• Different solutions, tailored to each platform

• Focus: preservation of the semantics!

• This talk: modular code generation
Synchronous block diagrams

• Fundamental model behind (discrete-time) **Simulink**, or **SCADE**
• Also very close to synchronous languages: Lustre, Esterel, …
Hierarchy
Hierarchy

Fundamental modularity concept
Code generation

• Generate code (in C, C++, Java, …) that implements the semantics

P.step( in ) returns out {
    tmp := A.step ( in );
    out := B.step ( tmp );
    return out;
}

• Code may be used for simulation or embedded control
  – Cf. Real-Time Workshop™
Modular code generation

- Code should be independent from context:
- Enables component-based design
- Takes care of IP issues
- Cf. AUTOSAR

Will P be connected like this?

...or like that?

...or like that?
Problem: “monolithic” code

False I/O dependencies

\[
P.\text{step}(x_1, x_2) \text{ returns } (y_1, y_2) \{
    y_1 := A.\text{step}(x_1);
    y_2 := B.\text{step}(x_2);
    \text{return } (y_1, y_2);
\}
\]
How common is this in practice?

True in all examples we tried

Engine control model in Simulink
Code generation – state of the art

- Either restrict diagram:
  - Break cycles at each level with **unit-delays** (c.f. SCADE)

- Or flatten (c.f. Simulink)
  - Remove diagram hierarchy
  - Check for dependency cycles
  - If none, generate code
  - Otherwise, reject diagram

- Non-modular!
Other approaches

• **Dynamic fix-point computation** [Edwards-Lee’03]:
  – Start with “bottom” (undefined value) assigned to all wires in the diagram
  – Keep calling “step()” functions until you find a fix-point
  – Hope for the best:
    • The fix-point may still contain “bottom” values
  – Unacceptable for safety-critical software

• Could check whether diagram is **constructive** [Malik’94, Berry et al.’96]
  – Expensive
  – Needs semantic information:
    • What is the function that this block computes?
    • Contrary to our black-box view
Our solution [DATE’08, RTAS’08, POPL’09]

• A general solution to the problem
  – No more flattening
  – No restrictions: handles all diagrams that can be handled by flattening

• A set of modular code generation algorithms
  – Some give more modular code than others
  – Notion of modularity is quantified
  – Exposes two fundamental trade-offs:
    Modularity vs. Reusability
    Modularity vs. Code size

• Optimality results
  – How to generate an optimal (minimal) interface

• Complexity results
  – Some problems are polynomial, some NP-complete
How do we do it?

• Generate for each block a **PROFILE = INTERFACE**
• Interface may contain **MANY** functions

```java
class P {
  public P.step1( in1 ) returns out1;
  public P.step2( in2 ) returns out2;

  P.step1( in1 ) {
    return A.step( in1 );
  }

  P.step2( in2 ) {
    return B.step( in2 );
  }
}
```
How do we do it?

class P {
    public P.step1( in1 ) returns out1;
    public P.step2( in2 ) returns out2;

    P.step1( in1 ) {
        return A.step( in1 );
    }

    P.step2( in2 ) {
        return B.step( in2 );
    }
}

A

B

P
How do we do it?

The function call order depends on the usage of the block!
Unit-delay blocks

• Memory element (register):

• One interface function is not enough:
Profile dependency graphs

- Profile also includes a DEPENDENCY GRAPH
- Encodes interface usage constraints

```java
class UnitDelay {
  public step( in ) returns void;
  public get( ) returns out;
  private state;

  step( in ) {
    state := in;
  }

  get( ) {
    return state;
  }
}
```
Overall method: example

A macro block $P$ and its internal diagram

<table>
<thead>
<tr>
<th>Interface functions</th>
<th>Profile dependency graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profile of $A$: (combinational)</td>
<td>A.step(x) returns y;</td>
</tr>
<tr>
<td></td>
<td>$x \rightarrow A$.step() $\rightarrow y$</td>
</tr>
<tr>
<td>Profile of $U$: (Moore-sequential)</td>
<td>U.get() returns y; U.step(x) returns void;</td>
</tr>
<tr>
<td></td>
<td>$y \rightarrow U$.get() $\rightarrow U$.step() $\rightarrow x$</td>
</tr>
<tr>
<td>Profile of $C$: (combinational)</td>
<td>C.step(x) returns y;</td>
</tr>
<tr>
<td></td>
<td>$x \rightarrow C$.step() $\rightarrow y$</td>
</tr>
</tbody>
</table>

SDG of $P$

SDG of $P$ clustered in two sub-graphs

Resulting interface functions and PDG of $P$
SDGs and clustering

Scheduling Dependency Graph (SDG) = composition of PDGs of sub-blocks

different clusterings = different tradeoffs
Trade-off: modularity vs. reusability

modularity becomes quantifiable

If block has N outputs then maximal reusability can be achieved with <= N+1 functions (tight)

Modularity-optimal method to achieve maximal reusability

---

class P {
    public Pstep( in1, in2 ) returns out1, out2;
    Pstep( in1, in2 ) {
        return (Astep( in1 ), Bstep( in2 ) );
    }
}

---

class P {
    public Pstep1( in1 ) returns out1;
    public Pstep2( in2 ) returns out2;
    Pstep1( in1 ) {
        return Astep( in1 );
    }
    Pstep2( in2 ) {
        return Bstep( in2 );
    }
}
An abstraction-oriented view

• Interface = an **abstraction** of the block

• It is a **conservative** abstraction
  – All original I/O dependencies are kept
  – More dependencies may be added
  – If no cycle occurs when using the interface, then no cycle would occur if instead we had flattened the block

• The **most conservative** abstraction is 1 function:
  – “step” function: computes outputs and updates state
  – Every output depends on all inputs

• An **exact** abstraction always exists
  – The set of I/O dependencies is finite
How to achieve optimality: overlapping clusters

A macro block $P$

2 outputs, 2 interface functions (optimal)
Overlapping clusters

=> code replication

\[
P.get1( x1, x2 ) \text{ returns } y1 \{
    \text{if (cA = 0) } \{
        (z1, z2) := A.step( x2 );
    \}
    \text{cA := (cA + 1) modulo 2;
    y1 := B.step( x1, z1 );
    return y1;
\}
\]

\[
P.get2( x2, x3 ) \text{ returns } y2 \{
    \text{if (cA = 0) } \{
        (z1, z2) := A.step( x2 );
    \}
    \text{cA := (cA + 1) modulo 2;
    y2 := C.step( z2, x3 );
    return y2;
\}
\]
Overlapping clusters
=> code replication

P.get1( x1, x2 ) returns y1 {
    if (cA = 0) {
        (z1, z2) := A.step( x2 );
    }
    cA := (cA + 1) modulo 2;
    y1 := B.step( x1, z1 );
    return y1;
}

P.get2( x2, x3 ) returns y2 {
    if (cA = 0) {
        (z1, z2) := A.step( x2 );
    }
    cA := (cA + 1) modulo 2;
    y2 := C.step( z2, x3 );
    return y2;
}
Another trade-off: modularity vs. code size

A macro block $P$

$\begin{align*}
  x_1 &\rightarrow P \\
  x_2 &\rightarrow A \\
  x_3 &\rightarrow C
\end{align*}$

$\begin{align*}
  y_1 &\rightarrow B \\
  y_2 &\rightarrow C
\end{align*}$

minimize code size $\Rightarrow$
non-overlapping (disjoint) clustering

- non-optimal in general
- optimal for disjoint clustering

Optimal disjoint clustering: NP-complete
But it can be reduced to sequence of SAT problems:
efficient in practice

With Christian Szegedy
Extension to triggered and timed diagrams

- Triggers and time:
  - Both concepts found in Simulink, SCADE, synchronous languages, …

Simulink/Stateflow diagram

Sample time

Triggered block
multi-rate models:

- B executed only when trigger = true
- All signals “present” always
- But not all updated at the same time
- E.g., output of B updated only when trigger is true

need initial value in case trigger = false at t = 0
Trigger elimination
Trigger elimination: atomic blocks

(a) eliminating the trigger from a combinational atomic block

(b) eliminating the trigger from a unit-delay
Trigger elimination: summary

• Can be done, efficiently

• But it destroys modularity:
  – Must propagate triggers top-down => “open the boxes”

• Solution:
  – Handle triggers directly, without eliminating them
Handling triggered diagrams directly

Scheduling Dependency Graph of P:

- Dependency added because of trigger
Timed diagrams

“static” multi-rate models

“TIMED” BLOCKS

(period, phase) specifications

A (3,1) → B (2,0) → C

P
Timed diagrams = “static” triggered diagrams

\[
\begin{align*}
A & \quad (3,1) \quad B \quad (2,0) \quad C \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad P
\end{align*}
\]

\[
\begin{align*}
A & \quad (3,1) \quad B \quad (2,0) \quad C \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad P
\end{align*}
\]

where

\[
(2,0)
\]

produces: true, false, true, false, …
Handling timed diagrams

• Could treat them as triggered diagrams

• But we can do better:

• Exploit the static information that timed diagrams provide:
  – To identify cases of false dependencies => accept more diagrams
  – To avoid firing blocks unnecessarily => more efficient code
Identifying false dependencies

A and B are never active at the same time

=>

Both dependencies are false
Eliminating redundant firings

Q: how often should P be fired?

**Simple answer:** every $\text{GCD}(5,2) = 1$ time unit = at every “clock cycle”

**Better answer:** at cycles $\{0,2,4,5,6,8,10, \ldots\} = \text{only when it needs to}$

**Problem:** (period,phase) representation not closed under union

**Solution:** Firing Time Automata
Firing Time Automata

\[ A \]

\[ B \]

\[ A \cup B \]
FTA division and multiplication

A
(4,0)

B
(2,0)

P

A
(2,0)

B
(1,0)

A \cup B

A \ntriangleleft A \cup B

B \ntriangleleft A \cup B

39
Firing Time Automata Operations

\[
A \cup B = (S_A \times S_B, (s_0^A, s_0^B), \{(s_A, s_B) | s_A \in F_A \lor s_B \in F_B \}, T_{A\cup B})
\]

\[
T_{A\cup B} = \{(s_A, s_B) \rightarrow (s_A', s_B') | s_A \rightarrow s_A' \in T_A \land s_B \rightarrow s_B' \in T_B\}
\]

\[
B \odot A = (S_A \times S_B, (s_0^A, s_0^B), \{(s_A, s_B) | s_B \in F_B \}, T_{B\odot A})
\]

\[
T_{B\odot A} = \left\{(s_A, s_B) \xrightarrow{1} (s_A', s_B') | s_A \rightarrow s_A' \in T_A \land s_B \rightarrow s_B' \in T_B \land s_A \in F_A\right\} \cup \left\{(s_A, s_B) \xrightarrow{\varepsilon} (s_A', s_B') | s_A \rightarrow s_A' \in T_A \land s_B \rightarrow s_B' \in T_B \land s_A \notin F_A\right\}
\]

\[
A \odot B = (S_A \times S_B, (s_0^A, s_0^B), \{(s_A, s_B) | s_A \in F_A \land s_B \in F_B \}, T_{A\odot B})
\]

\[
T_{A\odot B} = \{(s_A, s_B) \rightarrow (s_A', s_B') | s_A \rightarrow s_A' \in T_A \land s_B \rightarrow s_B' \in T_B \land s_A \in F_A\} \cup \{(s_A, s_B) \rightarrow (s_A', s_B') | s_A \rightarrow s_A' \in T_A \land s_A \notin F_A\}
\]
Theorem 3.1. For all deterministic firing-time automata $A, B$:

1. $(A \cup B)$ and $(A \circ B)$ are also deterministic firing-time automata.
2. $\emptyset \circ A = A \circ \emptyset = \emptyset$ and $(\{1\})^* \circ A = A \circ (\{1\})^* = A$.
3. $\emptyset \circ A = \emptyset$ and $A \circ (\{1\})^* = A$.
4. If $L(A) \supseteq L(B)$ then
   
   $A \circ (B \circ A) \equiv B$

5. As a corollary, from the fact that $L(A \cup B) \supseteq L(B)$, we get:

   $(A \cup B) \circ (B \circ (A \cup B)) \equiv B$
Firing Time Automata: summary

- Closed under union => can represent sets of firing times precisely

- Algebraic manipulation (“product”, “division”)

- Implemented as simple counters + set of accepting states

- Efficient code:
  - Fire a block only when we have to
Tool and experiments

- Tool implemented in Java
- Three clustering methods:
  - “step-get”: 1 or 2 clusters
  - “dynamic”: minimum no. clusters with overlapping
  - “ODC”: optimal disjoint clustering (uses SAT solving)
- Experiments:
  - Examples from Simulink’s demo suite, plus two from industrial partners
- Experimental results:

<table>
<thead>
<tr>
<th>model name</th>
<th>no. blocks</th>
<th>max no. outputs</th>
<th>max no. sub-blocks</th>
<th>total no. intf. func.</th>
<th>total code size (LOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
<td>macro</td>
<td>C,NS,MS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABS</td>
<td>27</td>
<td>3</td>
<td>1,0,2</td>
<td>4</td>
<td>57</td>
</tr>
<tr>
<td>Autotrans</td>
<td>42</td>
<td>9</td>
<td>4,0,5</td>
<td>11</td>
<td>108</td>
</tr>
<tr>
<td>Climate</td>
<td>65</td>
<td>10</td>
<td>4,0,6</td>
<td>29</td>
<td>144</td>
</tr>
<tr>
<td>Engine1</td>
<td>55</td>
<td>11</td>
<td>2,1,8</td>
<td>12</td>
<td>132</td>
</tr>
<tr>
<td>Engine2</td>
<td>73</td>
<td>13</td>
<td>3,2,8</td>
<td>13</td>
<td>182</td>
</tr>
<tr>
<td>Power window</td>
<td>75</td>
<td>14</td>
<td>6,2,6</td>
<td>11</td>
<td>180</td>
</tr>
<tr>
<td>X1</td>
<td>82</td>
<td>16</td>
<td>2,5,9</td>
<td>14</td>
<td>182</td>
</tr>
<tr>
<td>X2</td>
<td>112</td>
<td>16</td>
<td>7,9,0</td>
<td>14</td>
<td>261</td>
</tr>
</tbody>
</table>

[POPL’09]
It’s for real!

Diagram in our DATE’08 paper

Diagram in one of Simulink demos (engine control)

They are isomorphic!
Conclusions

- **Modular code generation framework**
  - No more flattening, no more IP issues, no restrictions on input
  - Handles triggered and timed diagrams

- Spectrum of methods

- Exposed fundamental trade-offs:
  - modularity vs. reusability
  - modularity vs. code size

- Optimality and complexity results

- Prototype tool and experiments
Thank you

• Questions ?