Modular code generation from synchronous models: modularity vs. reusability vs. code size

Stavros Tripakis

Joint work with
Roberto Lublinerman, Penn State
Semantics-preserving implementation of “high-level” models
Synchronous block diagrams

- Fundamental model behind (discrete-time) Simulink, SCADE, synchronous languages (Lustre, Esterel, …)
- Widely used in embedded systems
- **Synchronous**, deterministic semantics:

  ![Synchronous block diagram](image)

  - Throttle → Engine → Torque Converter → Gearset and Shift Mechanism → Transmission Control Unit → Vehicle Dynamics → mph
  - Transmission
  - brake

  ![Simulation and modeling diagram](image)
Hierarchy
Hierarchy

Fundamental modularity concept
Code generation

• Generate imperative 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, embedded control (“X-by-wire”), …
  – SCADE
  – Real-Time Workshop
  – …
Separate compilation

We want to do the same for synchronous block diagrams
Modular code generation

- Goal: generate code for a given block P
- Code should be independent from context:
  - Enables component-based design (c.f., AUTOSAR)

Will P be connected like this?

...or like that?

- Enables component-based design (c.f., AUTOSAR)
Problem with current approaches: “monolithic” code

False I/O dependencies =>
code not usable in some contexts

\[
P.\text{step}(x_1, x_2) \text{ returns } (y_1, y_2) \\
\{ \\
\quad y_1 := A.\text{step}( x_1 ); \\
\quad y_2 := B.\text{step}( x_2 ); \\
\quad \text{return } (y_1, y_2); \\
\}\]
Code generation – state of the art

- Either restrict diagram:
  - Break cycles at each level with **unit-delays** (SCADE)
  
- Or flatten (Simulink/RTW)
  - Remove diagram hierarchy

- Problem sometimes claimed impossible to solve [Girault’05]
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
    • There is a unique fix-point but it may contain “bottom” values
  – One would like to check that the fix-point does not contain “bottom” values

• Can do this by checking whether diagram is constructive [Malik’94, Berry et al.’96]
  – Undecidable in general, expensive otherwise
  – Needs semantic information:
    • What is the function that this block computes?
    • Contrary to our black-box component view
Our solution

• Modular:
  – No more flattening

• General:
  – No restrictions: handles all diagrams that can be handled by flattening

• Not one, but many solutions:
  – Explore different trade-offs
How do we do it?

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

```plaintext
P.step1( in1 ) returns out1 {
  return A.step( in1 );
}

P.step2( in2 ) returns out2 {
  return B.step( in2 );
}
```
How do we do it?

P.step1( in1 ) returns out1 {
    return A.step( in1 );
}

P.step2( in2 ) returns out2 {
    return B.step( in2 );
}
How do we do it?

The function call order depends on the usage of the block

```java
P.step1( in1 ) returns out1 {
    return A.step( in1 );
}

P.step2( in2 ) returns out2 {
    return B.step( in2 );
}
```
Modularity vs. Reusability

More modular, less reusable

More reusable, less modular

Modularity crucial for:
(1) Scalability
(2) IP issues

Modularity becomes quantifiable!
Profile dependency graphs

- Profile = Interface functions + DEPENDENCY GRAPH
- Graph encodes interface usage constraints

```java
class UnitDelay {
    private state;

    step( in ) returns void {
        state := in;
    }

    get() returns out {
        return state;
    }
}
```
Overall method

Input 1

A macro block \( P \) and its internal diagram

Input 2

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

SDG of \( P \)

Clustering

SDG of \( P \) clustered in two sub-graphs

Resulting interface functions and PDG of \( P \)
Trade-offs

**different clusterings => different trade-offs**

Currently have 3 clustering algorithms:
- Step-get clustering: 1 or 2 methods per block (classic)
- Dynamic clustering
- Optimal disjoint clustering
Dynamic clustering

• Group outputs w.r.t. input dependencies
• For each group, compute transitive fan-in

• Achieves:
  – Maximal reusability: code can be used in ANY context
  – Optimal modularity: minimal number of interface functions
  – Bound: \( \leq N+1 \) functions
    • \( N \): number of block outputs
Optimal modularity
=> overlapping clusters

2 interface functions (optimal)
Overlapping clusters
=> extra code for “dynamic” scheduling

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;
}
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;
}

• Code size crucial for embedded systems
• We want to minimize it =>
• We want disjoint clusters
Optimal disjoint clustering

- **Optimal disjoint clustering** problem:
  - How to cluster/partition a DAG into a minimal number of disjoint clusters, without introducing new input-output dependencies

- Optimal disjoint clustering is NP-complete
  - *With Christian Szegedy (Cadence labs)*

- Good news:
  - Can be reduced to a SAT problem (for given # of clusters)
  - Very efficient in practice!

[POPL’09]
Another trade-off: modularity vs. code size

A macro block $P$

3 interface functions (sub-optimal)
Extension to triggered and timed diagrams

• Triggers: found in Simulink, SCADE, synchronous languages, ...

• Sample times = static, periodic triggers
Triggered diagrams

**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

**Question:** do triggers increase expressiveness?
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: preserves the semantics

• But:
  – It requires flattening => 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:

A step

B step

C step

dependency added because of trigger
Timed diagrams

“static” multi-rate models

(period, phase) specifications
Timed diagrams = “static” triggered diagrams

where 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 GCD(5,2) = 1 time unit = at every “clock cycle”

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

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

Solution: Firing Time Automata
Firing Time Automata

A

B

\( (3,2) \)

\( (2,1) \)

\( A \cup B \)
FTA division
FTA union, division, multiplication

\[
\begin{align*}
A \cup B &= (S_A \times S_B, (s_0^A, s_0^B), \{(s_A, s_B) | s_A \in F_A \lor s_B \in F_B\}, T_{A \cup B}) \\
T_{A \cup B} &= \{(s_A, s_B) \rightarrow (s'_A, s'_B) | s_A \rightarrow s'_A \in T_A \land s_B \rightarrow s'_B \in T_B\}
\end{align*}
\]

\[
\begin{align*}
B \odot A &= (S_A \times S_B, (s_0^A, s_0^B), \{(s_A, s_B) | s_B \in F_B\}, T_{B \odot A}) \\
T_{B \odot A} &= \{(s_A, s_B) \rightarrow (s'_A, s'_B) | s_A \rightarrow s'_A \in T_A \land s_B \rightarrow s'_B \in T_B \land s_A \in F_A\} \cup \\
&\{(s_A, s_B) \rightarrow (s'_A, s'_B) | s_A \rightarrow s'_A \in T_A \land s_B \rightarrow s'_B \in T_B \land s_A \notin F_A\}
\end{align*}
\]

\[
\begin{align*}
A \odot B &= (S_A \times S_B, (s_0^A, s_0^B), \{(s_A, s_B) | s_A \in F_A \land s_B \in F_B\}, T_{A \odot B}) \\
T_{A \odot B} &= \{(s_A, s_B) \rightarrow (s'_A, s'_B) | s_A \rightarrow s'_A \in T_A \land s_B \rightarrow s'_B \in T_B \land s_A \in F_A\} \cup \\
&\{(s_A, s_B) \rightarrow (s'_A, s_B) | s_A \rightarrow s'_A \in T_A \land s_A \notin F_A\}
\end{align*}
\]
Correctness of algebraic operations

**Theorem 3.1.** For all deterministic firing-time automata $A, B$:

1. $(A \cup B)$ and $(A \ominus B)$ are also deterministic firing-time automata.
2. $\emptyset \ominus A = A \ominus \emptyset = \emptyset$ and $\{1\}^* \ominus A = A \ominus \{1\}^* = A$.
3. $\emptyset \ominus A = \emptyset$ and $A \ominus \{1\}^* = A$.
4. If $L(A) \supseteq L(B)$ then
   \[ A \ominus (B \ominus A) \equiv B \]
5. As a corollary, from the fact that $L(A \cup B) \supseteq L(B)$, we get:
   \[ (A \cup B) \ominus (B \ominus (A \cup B)) \equiv B \]
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

- Implemented in Java by Roberto Lublinerman
- Three clustering methods: "step-get" (max. 2 functions), optimal modularity (over-lapping clusters), and optimal disjoint clustering (uses SAT)

![Diagram of the tool process]

- Simulink model (.mdl file)
- Interface library for basic blocks
- Modular Code Generator
- Interfaces for macro blocks
- Java code

*demo available!*
Demo
Demo

Transmission

remember this?

\[ P \]

\[ x_1 \rightarrow B \rightarrow y_1 \]

\[ x_2 \rightarrow A \rightarrow y_1 \]

\[ x_3 \rightarrow C \rightarrow y_2 \]

transmission ratio
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>
<th>max red.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
<td>macro</td>
<td>C,NS,MS</td>
<td></td>
<td>S-G</td>
<td>Dyn</td>
</tr>
<tr>
<td>ABS</td>
<td>27</td>
<td>3</td>
<td>1,0,2</td>
<td>13</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Autotrans</td>
<td>42</td>
<td>9</td>
<td>4,0,5</td>
<td>11</td>
<td>fails</td>
<td>13</td>
</tr>
<tr>
<td>Climate</td>
<td>65</td>
<td>10</td>
<td>4,0,6</td>
<td>29</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>Engine1</td>
<td>55</td>
<td>11</td>
<td>2,1,8</td>
<td>12</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Engine2</td>
<td>73</td>
<td>13</td>
<td>3,2,8</td>
<td>13</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Power window</td>
<td>75</td>
<td>14</td>
<td>6,2,6</td>
<td>11</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>X1</td>
<td>82</td>
<td>16</td>
<td>2,5,9</td>
<td>14</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>X2</td>
<td>112</td>
<td>16</td>
<td>7,9,0</td>
<td>14</td>
<td>22</td>
<td>24</td>
</tr>
</tbody>
</table>

• Code reduction up to 75% for some blocks
• Execution time: negligible
Conclusions

• Modular code generation from synchronous models
  – Long-standing problem, sometimes claimed impossible to solve

• General framework, multiple solutions
  – Fundamental trade-offs: modularity, reusability, code size

• Key ideas: abstraction and interfaces

• Optimality results

• Prototype implementation

• Extensions to triggered and timed diagrams
  – Enrich interface with additional information (timing)
Thank you

Questions?
References

  - http://www-verimag.imag.fr/~tripakis/papers/date08.pdf
- R. Lublinerman and S. Tripakis. *Modular Code Generation from Triggered and Timed Block Diagrams*, RTAS’08
  - http://www-verimag.imag.fr/~tripakis/papers/rtas08.pdf