Several tools allow automatic code generation from high-level control models:

- Simulink Real-Time Workshop (Mathworks)
- Scicos (Inria)
- Lustre/SCADE (Verimag/Esterel-Tecnologies)
- Targetlink (DSpace)
- ASCET (ETAS)

This presentation is mostly based on our experience of the Lustre/SCADE Simulink Gateway and Code Generator.
Model-Based Development

These tools open the way to the Model-Based development method

- Design and validation on models (Simulink, Scicos)
- Automatic deployment on a given architecture guaranteeing faithfulness

This method avoids the manual coding phase which is error-prone.

It places the computerised control field ahead with respect to other computing fields
Steps in Code Generation

We first focus here on single thread code generation.

• Type inference
• Clock inference
• Code organisation
• Equation sorting
• Optimisation
Type Inference

Simulink/Stateflow is a partially typed system:
The user needs not care about types
Yet the resulting code should be typed.

⇒ need for type inference

This is the same situation as in functional languages (OCaml, Haskell). We can apply here the techniques that have been studied there (Hindley-Milner):

• writing type equations

• solving them by unification algorithm
Starting points:

- Blocks have a signature imposing type relations on their inputs and outputs
- The user can impose types in some blocks
Simulink Types

\[ SimT = \{ \text{type variables} \} \cup SimNum \cup \{ \text{boolean} \} \]

\[ SimNum = \{ \text{numerical type variables} \} \cup \{ \text{double, single, int8, uint8, int16, unit16, int32, uint32} \} \]
Block Type Signatures

Constant_\(\alpha\) : \(\alpha, \alpha \in SimT\)

Adder : \(\alpha \times \cdots \times \alpha \rightarrow \alpha, \alpha \in SimNum\)

Gain : \(\alpha \rightarrow \alpha, \alpha \in SimNum\)

Relation : \(\alpha \times \alpha \rightarrow boolean, \alpha \in SimT\)

Switch : \(\alpha \times \beta \times \alpha \rightarrow \alpha, \alpha, \beta \in SimNum\)

Logical Operator : \(boolean \times \cdots \times boolean \rightarrow boolean\)

Discrete Transfer Function : \(double \rightarrow double\)

Zero-Order Hold, Unit Delay : \(\alpha \rightarrow \alpha, \alpha \in SimT\)

Data Type Converter_\(\alpha\) : \(\beta \rightarrow \alpha, \alpha, \beta \in SimT\)

InPort, OutPort : \(\alpha \rightarrow \alpha, \alpha \in SimT\)
Writing and Solving Type Equations

Example: the stable subsystem

\[ \frac{1}{z} : tu = tx \]
\[ \frac{1}{z} : tu = tx \]
\[ ty = boolean \]

How to solve this system of equations?
Writing and Solving Type Equations

Example: the stable subsystem

How to solve this system of equations?

Let’s eliminate the type variable $tu$

This amounts to unify $tx$ and $tx$ which unify trivially.

Thus we get

$$stable : tx \rightarrow boolean$$
Writing and Solving Type Equations

More involved: the monitor subsystem

\[
\begin{align*}
\text{In1} & \quad \text{Out1} \\
\text{stable} & \quad \text{v} \\
\frac{1}{z} & \quad \text{In1} \\
\text{stable1} & \quad \text{w} \\
\text{OR} & \quad \text{i} \\
\text{AND} & \quad \text{y} \\
\text{Out1} & \quad \text{z}
\end{align*}
\]
Writing and Solving Type Equations

More involved: the monitor subsystem

Type equations:

\[
\begin{align*}
\frac{1}{z} : & \quad tu = tx \\
\text{stable} : & \quad tv = \text{boolean} \\
\text{stable1} : & \quad tw = \text{boolean} \\
\text{OR} : & \quad tv = tw = tt = \text{boolean} \\
\text{AND} : & \quad tt = tz = ty = \text{boolean} \\
\frac{1}{z} : & \quad tz = ty \\
\end{align*}
\]
Writing and Solving Type Equations

More involved: the monitor subsystem

We get \( ty = boolean \) and thus:

\[
monitor : tx \rightarrow boolean
\]
Other Type Analysis

- Complex Signals

- Signal dimensions

Follow the same lines
Clock Inference

An important question: when should the implementation compute?

A new issue that doesn’t exist in classical computing.

Yet the same typing techniques can apply:

- writing clock equations
- solving them by unification
Simulink Clocks

- Periodic clocks defined by:
  a rational positive period $\pi \in Q^+$
  and a rational positive phase $\theta < \pi$

- Triggers, akin to boolean signals

- Clock variables
The GCD Rule

When a block has several inputs with different sample times, the output sample time is given by the generalised gcd rule:

\[
ggcd((\pi_1, \theta_1), (\pi_2, \theta_2)) = \begin{cases} 
(\pi = \gcd(\pi_1, \pi_2), \theta_1 \text{ modulo } \pi) & \text{if } \theta_1 = \theta_2 \\
(\gcd(\pi_1, \pi_2, \theta_1, \theta_2), 0) & \text{otherwise}
\end{cases}
\]
The GCD Rule

When a block has several inputs with different sample times, the output sample time is given by the generalised gcd rule:

\[
\text{ggcd}((2, 0), (3, 0)) = (\text{gcd}(2, 3), 0) = (1, 0)
\]
Useful Simplifications

- \( ggcd \) is associative commutative. So we can write:
  \[
  ggcd(a, b, c) = ggcd(a, ggcd(b, c))
  \]

- \( ggcd(a, a, b) = ggcd(a, b) \)

- The solution of \( a = ggcd(a, b) \) is \( a = b \)
Simulink Clock Signatures

Sources$^{-1}$ : $\alpha$

Sources$^{(\pi, \theta)}$ : $(\pi, \theta)$

Maths$^{-1}$ : $\alpha_1 \times \alpha_2 \ldots \times \alpha_n \rightarrow ggcd(\alpha_1, \alpha_2, \ldots \alpha_n)$

Maths$^{(\pi, \theta)}$ : $\alpha_1 \times \alpha_2 \ldots \times \alpha_n \rightarrow (\pi, \theta)$

Discrete$^{-1}$ : $\alpha \rightarrow \alpha$

Discrete$^{(\pi, \theta)}$ : $\alpha \rightarrow (\pi, \theta)$

Sinks$^{-1}$ : $\alpha \rightarrow \alpha$

Sinks$^{(\pi, \theta)}$ : $\alpha \rightarrow (\pi, \theta)$
The Triggered Subsystem Inference Rule

\[
\text{if } S : \alpha^m \rightarrow \alpha^n \\
\text{and } b : \beta \\
\text{then } S^b : b^m \rightarrow \beta^n
\]

where \( S^b \) means: the subsystem \( S \) triggered by the signal \( b \)

This amounts to saying that a triggered subsystem should have a single clock and that once triggered by a signal, it executes only when the trigger is active and then its outputs are held when the trigger is inactive.
The Triggered Subsystem Inference Rule

\[ \text{if } S : \alpha^m \rightarrow \alpha^n \]
\[ \text{and } b : \beta \]
\[ \text{then } S^\hat{b} : b^m \rightarrow \beta^n \]

where \( S^\hat{b} \) means: the subsystem \( S \) triggered by the signal \( b \)

This amounts to saying that a triggered subsystem should have a single clock and that once triggered by a signal, it executes only when the trigger is active and then its outputs are hold when the trigger is inactive.
Clocking the single thread case

Clock inference provides us with a basic clock

- either the fastest periodic clock

- or a clock variable, if there is no periodic clock in the design

In any case, every other clock of the system is slower and can be considered as a trigger.

In case of periodic clocks, these triggers are implicit and we must make them explicit by means of clock dividers and phasers.
Clocking the single thread case

Clock inference provides us with a basic clock

- either the fastest periodic clock
- or a clock variable, if there is no periodic clock in the design

In any case, every other clock of the system is slower and can be considered as a trigger.

In case of periodic clocks, these triggers are implicit and we must make them explicit by means of clock dividers and phasers.
Example: A (3,1) clock

\[
\begin{align*}
\text{1} & \rightarrow \text{In1} \rightarrow \text{Out1} \\
\text{Constant} & \rightarrow (3,1)\text{clock} & \text{Scope}
\end{align*}
\]

\[
\begin{align*}
\text{1} & \rightarrow \text{In1} \\
\text{1} & \rightarrow \text{Out1} \\
\text{z} & \rightarrow 1 \\
\text{Unit Delay} & \rightarrow \text{Switch} \\
\rightarrow & \text{Compare To Zero} \\
\rightarrow & \text{Integer Delay} \\
\rightarrow & \text{Out1}
\end{align*}
\]
Example: A (3,1) clock
From Subsystems to Difference Equations

A single-clock subsystem is a system of difference equations

\[ U(0), \]
\[ U(n + 1) = F(U(n), X(n)) \]
\[ Y(n) = G(U(n), X(n)) \]

where

- \( U \), vector of state variables
- \( X \), vector of inputs
- \( Y \), vector of outputs
- \( F \), state transition function,
- \( G \), output function,
The idea is to build on object-oriented programming ($C^{++}$, Java)

<table>
<thead>
<tr>
<th></th>
<th>Simulink</th>
<th>Object-Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>system of equations</td>
<td></td>
<td>class</td>
</tr>
<tr>
<td>state variables</td>
<td></td>
<td>class attributes</td>
</tr>
<tr>
<td>initialisation</td>
<td></td>
<td>object creation</td>
</tr>
<tr>
<td>functions</td>
<td></td>
<td>class methods</td>
</tr>
</tbody>
</table>
From Difference Equations to Programs

\[ U(0), \]
\[ Y(n) = G(U(n), X(n)) \]
\[ U(n + 1) = F(U(n), X(n)) \]

A local version of state variables is used so as to not depend on the order of computations.

class subsyst{
private:
    state u;
public:
    subsyst(state u_init){
        u = u_init;
    }
    output step(input x){
        state up;
        output y = g(u, x);
        up = f(u, x);
        u = up;
        return y;
    }
}

plan
\[ \text{def} \]
Example

\[ U_0(z) = z^{-1}(a_0X(z) - b_0Y(z)) \]
\[ U_1(z) = z^{-1}(a_1X(z) - b_1Y(z) + U_0(z)) \]
\[ Y(z) = a_2X(z) + U_1(z) \]
From Simulink to Difference Equations

\[
U_0(z) = z^{-1}(a_0 X(z) - b_0 Y(z)) \\
U_1(z) = z^{-1}(a_1 X(z) - b_1 Y(z) + U_0(z)) \\
Y(z) = a_2 X(z) + U_1(z)
\]

\[
U_0(n + 1) = a_0 X(n) - b_0 Y(n) \\
U_1(n + 1) = a_1 X(n) - b_1 Y(n) + U_0(n) \\
Y(n) = a_2 X(n) + U_1(n)
\]
From Difference Equations to Programs

\[ Y(n) = a_2 X(n) + U_1(n) \]
\[ U_0(n + 1) = a_0 X(n) - b_0 Y(n) \]
\[ U_1(n + 1) = a_1 X(n) - b_1 Y(n) + U_0(n) \]

```cpp
class second_order {
    private:
        double a0, a1, a2, b0, b1;
        double u0, u1;
    public:
        second_order(double u0, u1)
        {
            ....
        }
        double step(double x) {
            ...
            return y;
        }
    }
```
Building the step function

\[
U_0(n + 1) = a_0 X(n) - b_0 Y(n)
\]
\[
U_1(n + 1) = a_1 X(n) - b_1 Y(n) + U_0(n)
\]
\[
Y(n) = a_2 X(n) + U_1(n)
\]

```c
double step(double x) {
    double y, up0, up1;
    y = a2 * x + u1;
    up0 = a0 * x - b0 * y;
    up1 = a1 * x - b1 * y + u0;
    u0 = up0;
    u1 = up1;
    return y;
}
```
Optimisations

Many optimisations are possible, due to the equational semantics of block-diagrams

- elimination of unused buffers
- eliminating buffers by reordering computations
- modify the original system by $z^{-1}$ comutation
double step(double x) {
    double y, up0, up1;
    y = a2 * x + u1;
    up0 = a0 * x - b0 * y;
    up1 = a1 * x - b1 * y + u0;
    u0 = up0;
    u1 = up1;
    return y;
}

u1 is not used since up1 is computed
double step(double x) {
    double y, up0;
    y = a2*x + u1;
    up0 = a0*x - b0*y;
    u1 = a1*x - b1*y + u0;
    u0 = up0;
    return y;
}

ul depends on u0 but not the converse. We can compute ul first.
Commuting Delays

This is due to the property:

If $f$ a static function (not encompassing $z$ operators), then

$$zf(x, y) = f(zx, zy)$$

Note that $f$ can be non-linear, boolean,...

Known as “Leiserson & Saxe retiming”

Used in hardware generation, pipelining,...

Is this obvious ???
Optimisations

Many possibilities

But the problem of buffer optimisation is a “hard problem”

Heuristics are required and obtaining “good code” is not obvious

Yet preferable to human coding
Code generation is modular

class upper{
private:
    ...
public:
    output step(inputs...){
        ...
        y = some_function.step(x);
        ...
    }
}
Subsystems

Code generation is modular

But for false causality loops!
False Causality Loops

Is this a correct model?
What is the result?
False Causality Loops

There is no causality loop
But modular code generation is not possible
False Causality Loops

There is no causality loop
But modular code generation
is not possible
However code can be generated by inlining
False Causality Loops

Inlining

class upper{
public:
    upper(){}
    a_type step(a_type In1) {
        a_type double_id_In1 = In1;
        a_type double_id_Out1 = double_id_In1;
        a_type double_id_In2 = double_id_Out1;
        a_type double_id_Out2 = double_id_In2;
        a_type Out1 = double_id_Out2;
        return Out1;
    }
};
Inlining vs Modularity

Inlining allow generating code for more models

Yet modular code generation is often preferred:

- Better readability

- Easier testing
The outputs of triggered subsystems are *state variables* that have to be initialised properly.
Verification of Simulink/Stateflow through SCADE

Uses the translation to Lustre

- Through the Prover Plugin:
  Translation to TECLA the internal language of the Prover Engine
  Then proof by induction using the Prover SAT - solver.

- using Lustre tools
  for instance, enumerative model-checking through automata code generation.
Automaton Code Generation

A boolean Lustre program:

- \( U(n + 1) = F(U(n), X(n)) : \text{State} \)
- \( Y(n) = G(U(n), X(n)) : \text{Output} \)
- \( U(0) : \text{initial state} \)

Enumerate all the reachable states and for each reachable state and possible input value, the corresponding output value.
Application to code generation

Build the “verification program”:

If the program is purely boolean and satisfies the property, the code generation into automaton should never output the value “false”
Example

Does this program:

satisfies this property
node once(x: bool) returns (y: bool);
let y = x or (false -> pre y);
tel
node monitor(x: bool) returns(prop: bool);
let
prop = (stable(x) or stable(false -> pre x))
    and (true -> pre prop);
tel
Verification

We build the verification node

```plaintext
node verif(x: bool) returns (prop : bool);
var y: bool;

let prop = monitor(y);
    y = once (x);
tel
```

and we generate the automaton
switch(ctx->current_state){
case 0:
    ctx->_V2 = _true;
    verif_O_prop(ctx->client_data, ctx->_V2);
    if(ctx->_V1){
        ctx->current_state = 1; break;
    } else {
        ctx->current_state = 2; break;
    }

case 1:
    ctx->_V2 = _true;
    verif_O_prop(ctx->client_data, ctx->_V2);
    ctx->current_state = 3; break;

case 2:
    ctx->_V2 = _true;
    verif_O_prop(ctx->client_data, ctx->_V2);
    if(ctx->_V1){
        ctx->current_state = 1; break;
    } else {
        ctx->current_state = 2; break;
    }

case 3:
    ctx->_V2 = _true;
    verif_O_prop(ctx->client_data, ctx->_V2);
    ctx->current_state = 3; break;
}
}
Conclusions on Single-Thread Code Generation

It allows generating code for any discrete-time model that can be simulated.

Allows many optimisations

The need for Real-Time Operating System is minimised

Provides in general robust and efficient code