Verilog Synthesis

Summer School on Generative and Transformational Techniques in Software Engineering, 2005
Outline

1. A Quick Tutorial on TL
2. Overview of a Synthesis Problem
3. Design and Implementation
The Signature of a Rewrite Rule

Definition

From the perspective of type, a labelled rewrite rule has the form:

\[ id : \text{pattern} \rightarrow s^n [ \text{if Boolean} ] \]

- the type \(id\) is the set of identifiers
- the type \(\text{pattern}\) is the set of \text{parse expressions} over a given grammar
- the type \(s^n\) is the set of \text{strategies} of order \(n\)
- the Boolean condition is optional
The Signature of a Rewrite Rule

Definition

From the perspective of type, a labelled rewrite rule has the form:

\[ id : \text{pattern} \rightarrow s^n [ \text{if Boolean} ] \]

- the type \text{id} is the set of identifiers
- the type \text{pattern} is the set of \text{parse expressions} over a given grammar
- the type \( s^n \) is the set of \text{strategies} of order \( n \)
- the Boolean condition is optional
A **parse expression** is a notation for describing parse trees (concrete syntax trees). Parse expressions are of type **pattern**.

Let \( G = (N, T, P, S) \) denote a context-free grammar.

- \( A_{id} \) is a parse expression if \( A \in N \). In the context of matching, the parse expression \( A_{id} \) is a variable quantified over the set \( \{ \alpha \mid A \xrightarrow{*}_{G} \alpha \land \alpha \in T^* \} \).

- \( A[\alpha'] \) is a parse expression if \( A \xrightarrow{+}_{G} \alpha \). The parse expression \( A[\alpha'] \) is quantified over the set \( \{ \beta \mid A \xrightarrow{+}_{G} \alpha \xrightarrow{*}_{G} \beta \land \beta \in T^* \} \).
A parse expression is a notation for describing parse trees (concrete syntax trees). Parse expressions are of type `pattern`.

- Let $G = (N, T, P, S)$ denote a context-free grammar.
  - $A_{id}$ is a parse expression if $A \in N$. In the context of matching, the parse expression $A_{id}$ is a variable quantified over the set $\{\alpha \mid A \xrightarrow{\ast}_G \alpha \land \alpha \in T^*\}$
  - $A[\alpha']$ is a parse expression if $A \xrightarrow{+}_G \alpha$. The parse expression $A[\alpha']$ is quantified over the set $\{\beta \mid A \xrightarrow{+}_G \alpha \xrightarrow{\ast}_G \beta \land \beta \in T^*\}$
A Quick Tutorial on TL
Overview of a Synthesis Problem
Design and Implementation

Parse Expression Examples

stmtS ::= stmt stmtS | ()
stmt ::= blocking_assign ";" | par_block | ...
par_block ::= "fork" stmtS "join"
blocking_assign ::= lvalue "=" E
lvalue ::= ...
E ::= id | ...
...

stmtS_1
stmtS[stmt_1 stmtS_1]
stmtS[blocking_assign_1; stmtS_1]
lvalue_1 = E_1; stmtS_1
lvalue_1 = E_1;
stmtS[stmt_1 stmt_2 stmt_3 stmt_4]
A strategic expression is an expression whose evaluation yields a strategy. A strategic expression of order $n$ has type $s^n$.

- $s^0$ = a pattern
- $s^{n+1} = lhs \rightarrow s^n$
- $s^1$ is a first-order strategy
- $s^2$ is a second-order strategy
- $s^n$ where $n > 1$ is a higher-order strategy
- the result of applying a strategy of type $s^{n+1}$ to a tree $t$ is a strategy of type $s^n$
A strategic expression is an expression whose evaluation yields a strategy. A strategic expression of order $n$ has type $s^n$.

- $s^0$ is a pattern
- $s^{n+1} = lhs \rightarrow s^n$
- $s^1$ is a first-order strategy
- $s^2$ is a second-order strategy
- $s^n$ where $n > 1$ is a higher-order strategy
- the result of applying a strategy of type $s^{n+1}$ to a tree $t$ is a strategy of type $s^n$
An Example of a First-Order Strategy

\[
\text{wrap: } \text{stmt[ blocking_assign}_1; ] } \rightarrow \text{stmt[ fork blocking_assign}_1; \text{ join ]}
\]

\[
\begin{align*}
stmtS & ::= \text{stmt stmtS | ()} \\
\text{stmt} & ::= \text{blocking_assign ";" | par_block | ...} \\
\text{par_block} & ::= \text{"fork" stmtS "join"} \\
\text{blocking_assign} & ::= \text{lvalue "=" E} \\
\text{lvalue} & ::= \text{...} \\
\text{E} & ::= \text{id | ...} \\
\ldots & 
\end{align*}
\]
propagate: blocking_assign[ id₁ = E₁ ] → E[ id₁ ] → E₁

stmtS ::= stmt stmtS | ()
stmt ::= blocking_assign “;” | par_block | ...
par_block ::= “fork” stmtS “join”
blocking_assign ::= lvalue “=” E
lvalue ::= ...
E ::= id | ...
...
**Definition**

**Combinators** are operators that can be used to construct strategies.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>«+</td>
<td>left-biased choice</td>
<td>$s_1 «+ s_2$</td>
</tr>
<tr>
<td>‡&gt;</td>
<td>right-biased choice</td>
<td>$s_1 ‡&gt; s_2$</td>
</tr>
<tr>
<td>«;</td>
<td>left-to-right sequential composition</td>
<td>$s_1 «; s_2$</td>
</tr>
<tr>
<td>»;</td>
<td>right-to-left sequential composition</td>
<td>$s_1 »; s_2$</td>
</tr>
</tbody>
</table>

*transient* a unary combinator

*hide* a unary combinator
A Bottom-up Left-to-right (BUL) Generic Traversal
A Top-down Left-to-right (TDL) Generic Traversal
TDL Traversal from a Strategic Perspective
TD Traversal from a Strategic Perspective
First-Order Strategy Application

\[ S^1 \]

Verilog Synthesis
Definitions of Some First-Order Traversals

\[
\begin{align*}
\text{def BUL } s & = \text{all_thread_left(BUL\{s\}) } <; \ s \\
\text{def TDL } s & = s <; \text{all_thread_left(TDL\{s\})} \\
\text{def Special_TD } s & = (\text{par_block}_1 \rightarrow \text{TDL}\{s\}(\text{par_block}_1)) <+ \\
& \text{all_broadcast(Special_TD}\{s\})
\end{align*}
\]
Higher-Order Strategy Application

\[ S_{n+1} \rightarrow S_1^n \]

\[ S_{n+1} \rightarrow S_2^n \]

\[ \vdots \]

\[ S_{n+1} \rightarrow S_m^n \]
Higher-Order Strategy Composition
## A Taxonomy of Some Generic Higher-Order Traversals

<table>
<thead>
<tr>
<th>Traversal</th>
<th>bottom-up</th>
<th>top-down</th>
<th>left-to-right</th>
<th>right-to-left</th>
<th>⊕</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcond_tdl</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>rcond_tdr</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>+</td>
</tr>
<tr>
<td>lcond_tdl</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>⇓</td>
</tr>
<tr>
<td>lcond_tdr</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>⇓</td>
</tr>
<tr>
<td>rcond_bul</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>rcond_bur</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>+</td>
</tr>
<tr>
<td>lcond_bul</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>⇓</td>
</tr>
<tr>
<td>lcond_bur</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>⇓</td>
</tr>
<tr>
<td>lseq_tdl</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>;</td>
</tr>
<tr>
<td>lseq_tdr</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>;</td>
</tr>
<tr>
<td>lseq_bul</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>;</td>
</tr>
<tr>
<td>lseq_bur</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>;</td>
</tr>
</tbody>
</table>
Verilog Synthesis
An Overview of Verilog

- Verilog is a hardware description language (HDL)
- Verilog has a C-like syntax
- Verilog has constructs to describe parallel computation and sequential computation
  - The items in a module execute in parallel (e.g., `continuous assignment` statements and `always` statements)
  - Blocks of the form `[begin ... end]` execute the statements in their bodies in sequential order
  - Blocks of the form `[fork ... join]` execute the statements in their bodies in parallel
Verilog is a hardware description language (HDL)

Verilog has a C-like syntax

Verilog has constructs to describe parallel computation and sequential computation

- The items in a module execute in parallel (e.g., `continuous assignment` statements and `always` statements)
- Blocks of the form `[begin ... end]` execute the statements in their bodies in sequential order
- Blocks of the form `[fork ... join]` execute the statements in their bodies in parallel
An Overview of Verilog

- Verilog is a hardware description language (HDL)
- Verilog has a C-like syntax
- Verilog has constructs to describe parallel computation and sequential computation
  - The items in a module execute in parallel (e.g., `continuous assignment` statements and `always` statements)
  - Blocks of the form `begin ... end` execute the statements in their bodies in sequential order
  - Blocks of the form `fork ... join` execute the statements in their bodies in parallel
An Overview of Verilog

- Verilog is a hardware description language (HDL)
- Verilog has a C-like syntax
- Verilog has constructs to describe parallel computation and sequential computation
  - The items in a module execute in parallel (e.g., `continuous assignment` statements and `always` statements)
  - Blocks of the form `[begin ... end]` execute the statements in their bodies in sequential order
  - Blocks of the form `[fork ... join]` execute the statements in their bodies in parallel
An Overview of Verilog

- Verilog is a hardware description language (HDL)
- Verilog has a C-like syntax
- Verilog has constructs to describe parallel computation and sequential computation
  - The items in a module execute in parallel (e.g., *continuous assignment* statements and *always* statements)
  - Blocks of the form `begin ... end` execute the statements in their bodies in sequential order
  - Blocks of the form `fork ... join` execute the statements in their bodies in parallel
An Overview of Verilog

- Verilog is a hardware description language (HDL)
- Verilog has a C-like syntax
- Verilog has constructs to describe parallel computation and sequential computation
  - The items in a module execute in parallel (e.g., *continuous assignment* statements and *always* statements)
  - Blocks of the form `[begin ... end]` execute the statements in their bodies in sequential order
  - Blocks of the form `[fork ... join]` execute the statements in their bodies in parallel
Synthesis Goals

Goal

*Develop a transformation-based synthesis system that removes sequential computation from Verilog programs*

Goal

*Construct a transformation whose manipulations are guided by correctness-preserving algebraic laws*
Synthesis Goals

Goal

*Develop a transformation-based synthesis system that removes sequential computation from Verilog programs*

Goal

*Construct a transformation whose manipulations are guided by correctness-preserving algebraic laws*
module example (out1, out2, in, cs);
input in;
output out1, out2;
always@(*) begin
    out1 = !cs;
    ns = out1;
    out2 = !out1 || c2;
    if (cs == 0) out2 = !out1;
    else ns = 0;
end
endmodule
module example(out1, out2, in, cs);
input in;
output out1, out2;
always@(*) begin
    fork out1 = !cs; ns = ns; out2 = out2; join
    fork ns = out1; out1 = out1; out2 = out2; join
    fork out2 = !out1 || c2; out1 = out1; ns = ns; join
    if ( cs == 0 ) fork out2 = !out1; out1 = out1; ns = ns; join
    else fork ns = 0; out1 = out1; out2 = out2; join
end
endmodule
module example(out1, out2, in, cs);
input in;
output out1, out2;
always@(*) begin
    fork
        ns = (cs == 0) ? !cs : 0;
        out1 = (cs == 0) ? !cs : !cs;
        out2 = (cs == 0) ? !!cs : !!cs || c2;
    join
end
endmodule
Law

Parallel assignment completion.

\[(x, y, \ldots := e, f, \ldots) = (x, y, \ldots, z := e, f, \ldots, z)\]

Law

Parallel assignment reordering.

\[(x, \ldots, y, z, \ldots := e, \ldots, f, g, \ldots) = (x, \ldots, z, y, \ldots := e, \ldots, g, f, \ldots)\]
Law

**Parallel assignment constant propagation.**

\[
\left( \vec{v} := g; \vec{v} := h(\vec{v}) \right) = \left( \vec{v} := h(g) \right) \text{ where } \vec{v} \text{ is an assignment state.}
\]

Law

**Conditional constructor elimination.**

\[
\left( ((\vec{v} := g) \triangleleft c \triangleright (\vec{v} := h)) \right) = \left( \vec{v} := (g \triangleleft c \triangleright h) \right)
\]
modulee ::= module module_id “;” m_item_0orMore endmodule
m_item_0orMore ::= module_item m_item_0orMore | ()
module_item ::= continuous_assign | always_stmt | ...
continuous_assign ::= “assign” lvalue “=” E “;”
always_stmt ::= “always” stmt
stmtS ::= stmt stmtS | ()
stmt ::= blocking_assign “;” | seq_block | par_block | ...
seq_block ::= “begin” stmtS “end”
par_block ::= “fork” stmtS “join”
b-blocking_assign ::= lvalue “=” E
Transformations yielding Assignment Normal Form

\[ \text{synthesize: } \text{BUL}\{\text{wrap }<; \text{Law1 } \}\text{<; BUL}\{\text{Law3 }<; \text{Law4 }\}\text{ } \]

\[ \text{wrap: } \text{stmt[ blocking_assign}_1; ] } \rightarrow \text{ stmt[ fork blocking_assign}_1; \text{ join } ] \]
Law1: \[ \text{modulee}_0 \]
\[ \rightarrow \]
\[ \text{Special_TD\{ lseq_bul\{ make_total \}[\text{modulee}_0] \}(\text{modulee}_0) } \]

**make_total:** \[ \text{blocking_assign}[i_1 = E_1] \rightarrow \text{transient(check}[i_1] \leftrightarrow \text{add}[i_1]) \]

**check:** \[ i_1 \rightarrow \text{stmtS}[ i_1 = E_2 \; \text{stmtS}_3 ] \rightarrow \text{stmtS}[ i_1 = E_2 \; \text{stmtS}_3 ] \]

**add:** \[ i_1 \rightarrow \text{stmtS}[ ] \rightarrow \text{stmtS}[ i_1 = i_1; ] \]
Law3: \[ \text{stmtS}\left[ \text{par\_block}_1 \ \text{par\_block}_2 \right] \]
\[
\rightarrow \text{BUL}\left\{ \text{lseq\_tdl}\{\text{propagate}\}\left[ \text{par\_block}_1 \right]\right\}\left( \text{stmtS}\left[ \text{par\_block}_2 \right]\right) \]

propagate: \[ \text{blocking\_assign}\left[ \text{id}_1 = \text{E}_1 \right] \rightarrow \text{E}\left[ \text{id}_1 \right] \rightarrow \text{E}_1 \]
Law4: \[
\text{stmt}[\text{if (E}_1\text{) stmt}_1 \text{ else stmt}_2]
\rightarrow
\text{BUL}\{\text{lseq_bul}\{ \text{convert}[E_1] \}[\text{stmt}_1] \}(\text{stmt}_2)
\]

convert \[E_0\]
\[
\text{blocking_assign}[\text{id}_1 = E_1]
: \text{blocking_assign}[\text{id}_1 = E_2]
\rightarrow
\text{blocking_assign}[\text{id}_1 = (E_0) \text{? E}_1 : E_2]
\]
A Quick Tutorial on TL
Overview of a Synthesis Problem
Design and Implementation

Result Summary

\[
\begin{align*}
\text{out1} &= \neg cs; \\
\text{ns} &= \text{out1}; \\
\text{out2} &= \neg \text{out1} \mid\!\mid c2; \\
\text{if} \ (cs == 0) \ \text{out2} &= \neg \text{out1}; \\
\text{else} \ \text{ns} &= 0;
\end{align*}
\]

\[
\Rightarrow
\begin{align*}
\text{fork} \\
\text{ns} &= (cs == 0) \ ? \ \neg cs : 0; \\
\text{out1} &= (cs == 0) \ ? \ \neg cs : \neg cs; \\
\text{out2} &= (cs == 0) \ ? \ \neg\neg cs : \neg\neg cs \mid\!\mid c2; \\
\text{join}
\end{align*}
\]
For Further Reading I

- J. Kyoda and H. Jifeng
  Towards an Algebraic Synthesis of Verilog

- V. L. Winter and M. Subramaniam
  Dynamic Strategies, Transient Strategies, and the Distributed Data Problem.

- V. L. Winter
  Strategy Construction in the Higher-Order Framework of TL.
  *Electronic Notes in Theoretical Computer Science (ENTCS)*, 124(1), 2004