MESCAL LANGUAGES REFERENCE MANUAL

by

Scott Weber

Memorandum No. UCB/ERL M03/41

15 October 2003

MESCAL LANGUAGES
REFERENCE MANUAL

by

Scott Weber

Memorandum No. UCB/ERL M03/41

15 October 2003

ELECTRONICS RESEARCH LABORATORY

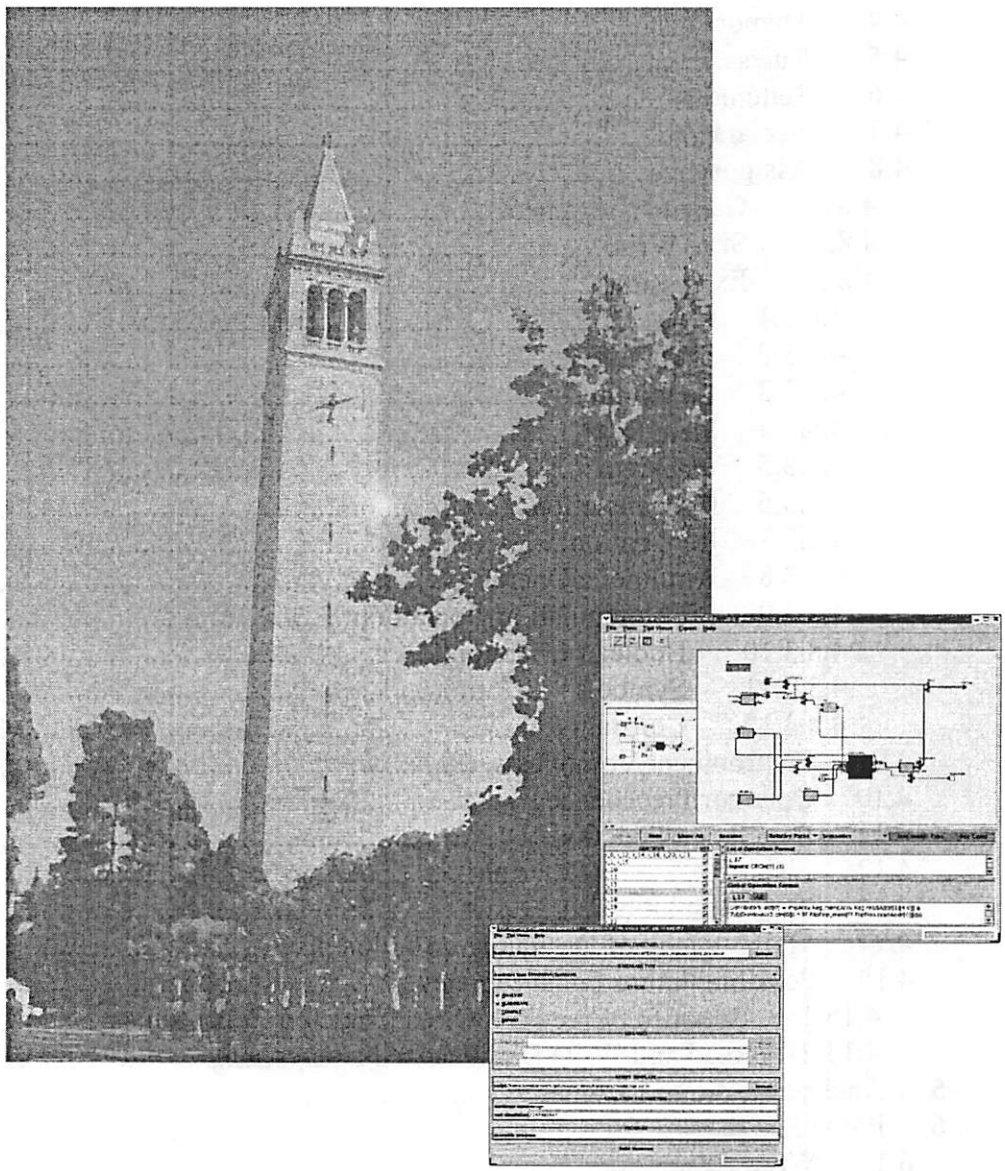
College of Engineering
University of California, Berkeley
94720

Mescal Languages Reference Manual

Scott Weber

Electronics Research Laboratory
University of California at Berkeley

sjweber@eecs.berkeley.edu



Contents

Revision History

Oct 15th, 2003	Initial Release

1 Introduction

The purpose of this document is to describe how one can design Tipi actors. There are three types of actors: analyzable actors, probe actors, and simulation only actors. Analyzable actors are used to create semantic operations on inputs, outputs, and state. These actors are synthesizable and can be translated to and simulated in C++ and Verilog. Probe actors are used to print out values in the design. Probe actors are not synthesizable, but can be translated to and simulated in C++ and Verilog. Simulation only actors can be used to embed arbitrary C++. These actors are restricted in how they can be used. These actors can be translated to and simulated in the C++ simulator. They cannot be translated to, simulated, or synthesized in Verilog.

The three types of actors are described in a constraint based polymorphic functional language. Constraints are specified as to how an actor can be used, and what happens when a particular constraint is met. Utilizing this methodology, one can create designs in a correct by construction manner. A design is exported as a set of primitive operations by analyzing the actors. Furthermore, a supporting set of tools including simulators, an assembler, and a Verilog hardware generator are generated for each design based on the primitive operations.

The languages coupled with the Tipi design environment provide a new architecture description language that is not limited to a particular family of architectures.

2 Port Type Language

The port type language is used to specify the types of ports on the actors. If a designer specifies the *_type* attribute on a port, then it overrides any type specified in the I/O section of the actor semantics language.

2.1 *Constants, Ports, and Instruction*

The terminals of the type expressions must either be constant numbers, a reference to the type of another port, or the special type *instruction* which represents the width of the instruction. The combination of these terminals along with the operators in the type expressions provides designers the ability to customize the semantics of their actors.

2.2 *Max*

It is important to note that a *max_type_expr* cannot be the second argument of a subtraction. This restriction is required in order to maintain the monoticity of the types.

The max expression returns the maximum width. For example, if the output port *out* of some actor is defined as:

```
max(inputA.type, inputB.type)
```

then the type of out is the type of *inputA* if *inputA* > *inputB*, otherwise, the type of *out* is the type of *inputB*.

2.3 *Add and Sub*

The operators + and – represent addition and subtraction, respectively.

2.4 *Log*

The operator *log* is defined as follows:

```
log(n), where n is a number          = 1    if n < 2
                                            log2(n) otherwise.
```

2.5 Ceil and Floor

The operators *ceil* and *floor* represent the ceiling and flooring operations, respectively.

The combination of *log* and *ceil* are particularly useful for describing address port types. For example, an address port defined as:

```
ceil(log(size))
```

will result in the appropriate type required to address a memory of size *size*.

The combination of *max*, *log*, and *floor* is useful for defining the width of a constant. For example, the type of a constant can be defined as:

```
max(out.type, floor(log(constant + 1)))
```

will produce a type for the constant that is big enough to represent the constant or it will produce a type that is larger and matches the type of the output *out*.

2.6 Converting to Integer

Since we have the log operator it is possible to have types with non-integer values. To deal with this we implicitly convert types to an integer using the *floor* operation. This means that the expression:

`log(3) + log(5)` is equivalent to `floor(log(3) + log(5))`.

3 Port Type Language Grammar

```
type                ::=   max_type_expr
                          |
                          max_type_expr + type
                          |
                          ( max_type_expr )
                          |
                          max_type_expr - type_expr
                          |
                          type_expr

max_type_expr       ::=   max ( max_type_expr_list )

max_type_expr_list  ::=   type_expr , max_type_expr_list
                          |
                          type_expr

type_expr           ::=   type_expr + type_expr
                          |
                          type_expr - type_expr
                          |
                          log ( type_expr )
                          |
                          ceil ( type_expr )
                          |
                          floor ( type_expr )
                          |
                          ( type_expr )
                          |
                          type_id

type_id             ::=   NUMBER
                          |
                          ID . type
                          |
                          @ ID . type
                          |
                          instruction . type
```

4 Analyzable Actor Language

The analyzable actor language is used to describe the computation components of the architectures. Any actor written in the actor semantics language can be simulated and translated into hardware.

4.1 *Comments*

C-style /* */ and C++-style // comments are supported.

4.2 *Actor Signature*

An actor is given a unique name followed by a list of its inputs and outputs. The semantics are then described in the program section. For example, an adder could have the signature:

```
add(inA, inB, out) { program }
```

4.3 *I/O List*

The list of inputs and outputs specified in the actor signature must be defined. Inputs are preceded by *input* and outputs are preceded by *output*. An optional type can be specified. If no type is specified or if the type cannot be statically resolved, then type resolution will attempt to resolve the type. For example, the inputs and outputs for the add actor could be defined as:

```
input inA;
input inB;
output <max(inA.type, inB.type)> out;
```

Multiports have the special syntax where the port name is preceded by a @. Some examples of input multiports (outputs can also be multiports) are shown:

```
input @din;             // A multiport with an undefined type
input <2> @din;         // A multiport with type 2
```

A multiport is a list of signals with the same type. A more in-depth discussion of lists is included later.

4.4 *Memory List*

Memories are prefixed by *reg* for registers and *ff* for flipflops. An optional type can be specified. If no type is specified or if the type cannot be statically resolved, then type resolution will attempt to resolve the type. A size for a memory can also be specified. If it is not then the memory is assumed to be of size 1. Some examples:

```
reg mem;              // 1 register with undefined type
reg <1> mem;          // 1 register with type 1
reg mem 10;           // 10 registers with undefined type
reg <width> mem 10;   // 10 registers with type width
ff mem;               // 1 flipflop with undefined type
ff <1> mem;           // 1 flipflop with type 1
ff mem 10;            // 10 flipflops with undefined type
ff <1+2> mem 10;      // 10 flipflops with type 3 (1+2)
```

4.5 *Rules*

The semantics of the actors are described as a product of sums formulation over rules[1]. Each rule contains a list of input and output ports. Each port in the rule can have one of six bindings. The bindings are as follows:

```
.p.1      The signal is present.
.p.0      The signal is not present.
.p.1.e    The signal is present and is an enumeration.
.p.0.e    The signal is not present and is an enumeration.
.p.1.op   The signal is present and is the instruction.
.p.0.op   The signal is not present and is the instruction.
```

The enumeration binding indicates that the signal takes on a set of enumerated values where a new unique value is created for each rule that contains the signal with the enumeration binding.

The instruction value indicates that the signal takes on the type of the instruction which is calculated after the control is generated.

In order for a rule to be true in the product of sums formulation, the signals on the ports in the rule's port list must match the bindings. The firing rules for the add are shown below:

```
or (
   fire (inA.p.1, inB.p.1, out.p.1)
      { out = inA + inB; }
   no_fire (inA.p.0, inB.p.0, out.p.0)
      { }
)
```

The firing rules are interpreted as: The actor is valid if *inputA*, *inputB*, and *out* are present or if *inputA*, *inputB*, and *out* are not present; if *inputA*, *inputB*, and *out* are present then *out* = *inputA* + *inputB*.

[1] In the future, we will support a quantifier-free first-order logic syntax.

The *and* operator can also be used to constrain the actors. Actors are only valid when the rule constraints are valid. A design is only valid when all actors are valid.

4.6 *Statements*

The contents of a rule are composed of set of statements that execute in parallel. Each statement contains an optional set of declarations followed by a set of parallel assignments. The structure of a rule is shown below:

```
statement1
    declaration
    assignment1
    …
    assignmentn
…
statementt
    declaration
    assignment1
    …
    assignmentm
```

4.7 *Declarations*

Each statement may contain an optional declaration section. The declarations can be used in the assignments of the statement. The structure of a statement with declarations is as follows:

```
let
    val decl1 = expr1;
    val decl2 = decl1;
    …
    val decls = exprs;
in
    assignment1
    …
    assignmentn
end
```

The declarations between *let* and *in* are performed sequentially so earlier declarations can be used in later ones. The assignments between *in* and *end* occur in parallel and can use any of the declarations.

4.8 *Assignments*

The set of parallel assignments consist of assignment expressions, state write expressions, and simulation expressions.

4.8.1 Output Assignments

Output assignments assign outputs to expressions on inputs and memory values. An additional constraint is added to the rule constraints so that each output is only assigned a single value. Assignment expressions have the form:

```
out = expr;
```

4.8.2 State Write

State writes are used to assign values to memories. State writes have the form:

```
mem[expr_addr] = expr_2;
```

State writes are non-blocking so the contents of the memory are updated at the end of the cycle.

4.8.3 Expressions

Each *expr* in the output assignments and state writes is composed of a combination of axiomatic operations. All examples are shown with output assignments, but the left hand side of the assigns can all be interchanged with mem[$expr_{addr}$].

4.8.3.1 Case

A case expression evaluates the first expression to determine what value to assign.

```
out = case (expr) { 0    => expr_0  |
                    ...
                  n-1  => expr_n-1 |
                  _    => expr_n
                  }
```

An ALU example using the case expression is shown below:

```
out = case (control){ 0 => inA + inB
                      1 => inA - inB
                      2 => inA * inB
                      3 => inA / inB
                      _ => inA
                    };
```

All case expressions must include a default case.

4.8.3.2 If-Then-Else

An if-then-else expression evaluates a Boolean expression *bool_expr* and then assigns the expression $expr_1$ in the then section if *bool_expr* evaluates to true; otherwise, the expression $expr_2$ in the else section is assigned. The format of an if-then-else expression is shown below:

```
out = if bool_expr then expr1 else expr2;
```

4.8.3.3 Conditionals

Conditional expressions provide an alternative syntax to the if-then-else expression. The format of a conditional expression is shown below:

```
out = bool_expr ? expr1 : expr2;
```

4.8.3.4 State Read

A state read expression is used to access a particular location in a memory. The format of a read expression is shown below:

```
out = mem[exprADDR];
```

4.8.3.5 Indexing

An indexing expression is used to extract a range of bits from any expression. The format of an indexing expression is shown below:

```
out = (expr)[type1:type2];
```

The indexing expression provides an alternative to creating declarations. For example, the two following statement using a declaration:

```
let
   val tmp = inA + inB;
in
   out = tmp[<max(inA.type, inB.type)> - 2:0];
end
```

is equivalent to the following statement using an indexed expression:

```
out = (inA + inB)[<max(inA.type, inB.type)> - 2:0];
```

The key difference between using a declaration and using an indexed expression is that the declaration is created once (an implementation would share the hardware to produce the declaration). In the case where the declaration is only used once, as in the example, the constructs will result in equivalent implementations.

4.8.3.6 Concatenation

The concatenation expression is used to construct a new signal by combining signals.

```
out = {inA, inB};
```

In this case, *out* is signal with the bits of *inA* followed by the bits of *inB*.

4.8.3.7 Replication

The replication expression provides a shorthand for concatenating the same signal a set number of times.

```
out = constant{inA};
```

In this case, *out* is signal with the bits of *inA* replicated *constant* number of times.

4.8.3.8 Arithmetic Operators

The standard arithmetic operators that are found in standard hardware description languages, such as Verilog and VHDL, and in standard high-level languages, such as C/C++ and Java, are supported. The types of these arithmetic operators can be found in the type system section. The supported operators are shown below:

```
out = expr1 + expr2;    // addition
out = expr1 - expr2;    // subtraction
out = expr1 * expr2;    // multiplication
out = expr1 / expr2;    // division
out = expr1 % expr2;    // modulo
```

4.8.3.9 Bit Manipulation Operators

The standard bit manipulation operators that are found in standard hardware description languages, such as Verilog and VHDL, are supported. The types of these bit manipulation operators can be found in the type system section. The supported operators are shown below:

```
out = expr1 & expr2;     // bit-wise and
out = expr1 ~& expr2;    // bit-wise nand
out = expr1 | expr2;     // bit-wise or
out = expr1 ~| expr2;    // bit-wise nor
out = expr1 ^ expr2;     // bit-wise xor
out = expr1 ~^ expr2;    // bit-wise xnor
out = expr1 ^~ expr2;    // bit-wise xnor
out = expr1 << expr2;    // logical shift left
out = expr1 >> expr2;    // logical shift right
out = ~expr;             // bit-wise invert
out = &expr;             // and reduce
out = ~&expr;            // nand reduce
out = |expr;             // or reduce
out = ~|expr;            // nor reduce
out = ^expr;             // xor reduce
out = ~^expr;            // xnor reduce
out = ^~expr;            // xnor reduce
```

4.8.3.10 Boolean Operators

The standard Boolean operators that are found in standard hardware description languages, such as Verilog and VHDL, and in standard high-level languages, such as C/C++ and Java, are supported. The types of these Boolean operators can be found in the type system section. The supported operators are shown below:

```
expr1 == expr2                  // equality
expr1 != expr2                  // inequality
expr1 < expr2                   // less than
expr1 <= expr2                  // less than or equal
expr1 > expr2                   // greater than
expr1 >= expr2                  // greater than or equal
bool_expr1 && bool_expr2        // Boolean and
bool_expr1 || bool_expr2        // Boolean or
!bool_expr                      // Boolean not
```

Currently, Boolean expressions can only be used within other expressions. It is not possible to assign an output a Boolean type, nor is it possible to create a Boolean type input. This does not mean you cannot perform Boolean operations; the bit manipulation operators on single bit unsigned integers can be used to do this. In the future, this restriction will be removed (see future enhancements).

4.8.3.11 Symbols

The symbol expressions are used to access the individual bits of the inputs and outputs of the actor. The following example shows how individual bits can be assigned and accessed:

```
out[30:0] = inA[31:1];
out[31]   = inA[0];
```

In the above case, the top 31 bits of *inA* are assigned to the lower 31 bits of *out*, and the upper bit of *out* is assigned the lower bit of *inA*. Also note, in the case of a single bit, only one range number is required (i.e. *inA[0]* is shorthand and equivalent to *inA[0:0]*).

The range operator can also be used to reverse bits in an assignment as shown below:

```
out = inA[0:32];
```

In the above case, *out* is assigned the bits of *inA* reversed.

In the examples above, the msb and lsb values in the ranges were all constants. The language also allows these to be arbitrary types as was shown in the indexing expression example. We could rewrite the first example in a polymorphic manner as:

```
out[out.type-2:0] = inA[out.type-1:1];
out[out.type-1]   = inA[0];
```

4.8.3.12 Constants

Constants are defined in a manner similar to Verilog constants. Examples of constants are shown below:

```
3'd7      // 3-bit decimal 7
3'D7      // 3-bit decimal 7
7         // decimal 7 (number of bits are unresolved)
6'o45     // 6-bit octal 45
6'O45     // 6-bit octal 45
4'he      // 4-bit hex e
4'He      // 4-bit hex e
3'b001    // 3-bit binary 001
3'B001    // 3-bit binary 001
```

The above examples demonstrates specifying constants when their widths are known. However, when the number of bits is not known we are allowed to specify our constants to be polymorphic. In any of the cases shown above, the number to the left of ' can be specified as a type expression. The definition of the constant actor as shown below takes advantage of this construct:

```
Const(out, trigger) {
    output out;
    input trigger;
    or (
        fire(trigger.p.1, out.p.1) {
            out =    <max(out.type,
                     floor(log(value) + 1)))'d value;
        }
        no_fire (trigger.p.0, out.p.0) {}
    )
}
```

In the above example, the width of the constant is either large enough to represent the environment variable *value* or it is 0-extended to be the width of *out*. If *out* is smaller than the number of bits required to represent *value*, then this actor is in an illegal context.

4.9 *Environment Variables*

As shown in the constant actor expression, environment variables that evaluate to constant unsigned integers are supported. These environment variables are specified outside the actor.

4.10 *Operator Precedence*

The order of operations follows that as defined in Verilog. The operator precedence is shown below. All the operators associate left to right.

unary operators:! & ~& | ~| ^ ~^ + ~ (highest precedence)

* / %

\+ -

<< >>

< <= > >=

== !=

& ~& ^ ~^

| ~|

&&

||

?: (lowest precedence).

If a different order is required then use parentheses.

4.11 *Simulation Functions*

In cases where the actor semantics language is not sufficient, simulation functions can be used. Simulation functions can be used for analysis, debugging, and interfacing to other programs. The simulation functions cannot change values in the simulation of the actor (i.e. they are read-only black boxed components). Extreme caution should be practiced when using simulation functions since they are not analyzed and could easily corrupt the generated simulators. Simulation functions are ignored during hardware generation. An example of a simulation function is shown below:

```
#begin sim
   print(inA) { cout << inA << endl; }
#end
```

Any simulation function must be prefixed by *#begin sim* and suffixed by *#end*. The body of the simulation function contains a name, a parameter list of ports, and an arbitrary C++ expression. A simulation function is translated into a C++ function. If the C++ is malformed, then it will only be caught during compilation of the simulator, not during the generation.

Simulation functions execute in parallel with the other assignments. This requires that one exercises caution when using simulation functions. For example if there are the two statements:

```
out = inA + inB;
#begin sim
   print(out) { cout << out << endl; }
#end
```

It is not clear what the value of *out* is since the statements are executed in parallel. If the value of *out = inA + inB* for the current cycle is desired then the following should be used:

```
out = inA + inB;
#begin sim
   print(inA, inB) { cout << (inA + inB) << endl; }
#end
```

Using a declaration also solves the problem:

```
let
   val tmp = inA + inB;
in
   out = tmp;
   #begin sim
      print(tmp) { cout << tmp << endl; }
   #end
end
```

A mechanism is available in the schematic editor to enable and disable the simulation functions within an actor or hierarchical actor. For actors, there is a Boolean parameter called *includeSimulation*. For hierarchical actors, there is a Boolean parameter called *includeSimulation* and *includeDescendantSimulation*. If *includeSimulation* is true for some hierarchical actor, then all of its descendants have their simulation functions enabled. If *includeSimulation* and *includeDescendantSimulation* are true for all ancestors of an actor, then that actor has its

simulation functions enabled. In all other scenarios, the simulation functions for the actor are disabled.

4.12 *Lists*

In order to better handle components that have the same basic semantic structure regardless on the number of input or output ports they have, support for lists has been implemented. The operations implemented only allow list manipulations in very controlled ways.

- The first list manipulation operation implemented allows for the generation of constraints based on the size of sets. The description of base memory actors utilizes this construct as is shown below:

```
Reg(readData, read, write, readAddr, writeAddr, writeData){

    output <writeData.type> @readData;
    input @read;
    input @write;
    input <ceil(log(size))> @readAddr;
    input <ceil(log(size))> @writeAddr;
    input <readData.type> @writeData;

    reg mem size;

    and(
        foreach(i) {
            or(
                _read(read[$i].p.1, readAddr[$i].p.1, readData[$i].p.1) {
                    readData[$i] = mem[readAddr[$i]];
                }
                _no_read(read[$i].p.0, readAddr[$i].p.0, readData[$i].p.0)
                    {}
            )
        }
        foreach(i) {
            or(
                _write(write[$i].p.1, writeAddr[$i].p.1, writeData[$i].p.1) {
                    mem[writeAddr[$i]] = writeData[$i];
                }
                _no_write(write[$i].p.0, writeAddr[$i].p.0, writeData[$i].p.0)
                    {}
            )
        }
    )
}
```

The above design is a polymorphic register file. The *foreach* expressions are interpreted with replication semantics. For each list that is indexed by the *foreach* variable (in this case *i*), we determine which list has a maximum cardinality N. The expression in the *foreach* expression is then replicated N times with *i* ranging from 0 to N-1. If we have a *readData* multiport (list) with 2 elements with a type of 32 bits, a *writeData* with 1 element, and a size of 32, then the following actor description gets expanded into the following. Type resolution is also shown.

```
Reg(readData, read, write, readAddr, writeAddr, writeData){

    output <32> readData[0];
    output <32> readData[1];
    input <0> read[0];
    input <0> read[1];
    input <0> write[0];
    input <5> readAddr[0];
    input <5> readAddr[1];
    input <5> writeAddr[0];
    input <32> writeData[0];

    reg <32> mem 32;

    and(
        or(
            _read_0(read[0].p.1, readAddr[0].p.1, readData[0].p.1) {
                readData[0] = mem[readAddr[0]];
            }
            _no_read_1(read[0].p.0, readAddr[0].p.0, readData[0].p.0)
                {}
        )
        or(
            _read_2(read[1].p.1, readAddr[1].p.1, readData[1].p.1) {
                readData[1] = mem[readAddr[1]];
            }
            _no_read_3(read[1].p.0, readAddr[1].p.0, readData[1].p.0)
                {}
        )
        or(
            _write_4(write[0].p.1, writeAddr[0].p.1, writeData[0].p.1) {
                mem[writeAddr[0]] = writeData[0];
            }
            _no_write_5(write[0].p.0, writeAddr[0].p.0, writeData[0].p.0)
                {}
            )
        )
    )
}
```

The resulting expansion yields a two read port, one write port 32-bit register file with 32 individual registers. To change the number of registers change *size*. To change the type, set the type of the read or write data port. To change the number of ports, simply add or remove relations to the read and write data ports. The resolved types of width 0 should be interpreted as these ports do not contain any information for the data path. This is because they are simply being used to constrain the control.

By default, an unconnected port is assumed to have one connected relation. This was chosen to stress the fact that *foreach* expressions only have replication semantics. If no relation was connected to the write data port in the above design, the same expansion would have occurred.

- The second list manipulation operation implemented allows for the generation of lists of input and output port bindings within a rule. The description of a polymorphic mux utilizes this construct as is shown below:

```
Mux(out, din, select) {

    output out;
    input <out.type> @din;
    input select;

    or(
        foreach(i) {
            fire(out.p.1, din[$i].p.1,
                foreach(j) { if ($i != $j) din[$j].p.0 },
                select.p.1.e) {
                    out = din[$i];
            }
        }
        no_fire(out.p.0, @din.p.0, select.p.0.e) {}
    )
}
```

The above describes a N:1 mux where N is the cardinality of the list on *din*. An enumerated binding has also been used here. This means that there is a unique value in the enumeration of *select* for each of the N *din* inputs. The nested *foreach* expression here allows for the selection of 1 of N signals. The only comparison allowed is inequality. The @ construct in the *no_fire* rule means that each element in the *din* list is bound to .p.0. If din was connected to 2 relations and had a width of 10, then the following expansion would occur. Type resolution is also shown.

```
Mux(out, din, select) {

    output <10> out;
    input <10> din[0];
    input <10> din[1];
    input <0> select;

    or(
        fire_0(out.p.1, din[0].p.1, din[1].p.0, select.p.1.e) {
            out = din[0];
        }
        fire_1(out.p.1, din[0].p.0, din[1].p.1, select.p.1.e) {
            out = din[1];
        }
        no_fire_2(out.p.0, din[0].p.0, din[1].p.0, select.p.0.e) {}
    )
}
```

Again, the select signal resolves to a width 0 signal since it only provides control constraints. Any other mux can be generated by connecting a different number of relations to *din* and by changing either the type of *din* or *out*. Furthermore, a 1:N demux has a similar construct except that the semantics of fire are *out[$i] = din*.

- The third list manipulation operation implemented allows for the generation of multiple assignments. The description of a polymorphic data demux utilizes this construct as is shown below:

```
DataDemux(out, din, select) {

    output @out;
    input <out.type> din;
    input select;

    or(
        fire(@out.p.1, din.p.1, select.p.1) {
            foreach(i) {
                out[$i] = case select of $i => din | _ => <out.type>'d0;
            }
        }
        no_fire(@out.p.0, din.p.0, select.p.0) {}
    )
}
```

The above construct allows for an arbitrary 1:N demux. A unique assignment is made to each element of *out*. If *out* was connected to 2 relations and had a type of 7 then the following expansion would occur. Type resolution is also shown.

```
DataDemux(out, din, select) {

    output <7> out[0];
    output <7> out[1];
    input <7> din;
    input <2> select;

    or (
        fire(out[0].p.1, out[1].p.1, din.p.1, select.p.1) {
            out[0] = case select of 0 => din | _ => 7'd0;
            out[1] = case select of 1 => din | _ => 7'd0;
        }
        no_fire(@out.p.0, din.p.0, select.p.0) {}
    )
}
```

In the case of data demux, the select signal is actually used in the data path, so it has a width (in this case a width of 1).

- The final list manipulation operation implemented allows for the generation of multiple match assignments within a case expression. The description of a polymorphic data mux utilizes this construct as is shown below:

```
DataMux(out, din, select) {

    output out;
    input <out.type> @din;
    input select;

    or(
        fire(out.p.1, @din.p.1, select.p.1) {
            out = case select of
            foreach(i) {
                $i => din[$i] |
            }
            _ => <out.type>'d0;
        }
        no_fire(out.p.0, @din.p.0, select.p.0) {}
    )
}
```

The above description allows for an arbitrary N:1 data mux. If *din* was connected 2 relations and had a width of 66 then the following expansion would occur. Type resolution is also shown.

```
DataMux(out, din, select) {

    output <66> out;
    input <66> din[0];
    input <66> din[1];
    input <1> select;

    or(
        fire(out.p.1, din[0].p.1, din[1].p.1, select.p.1) {
            out = case select of  0 => din[0] |
                                  1 => din[1] |
                                  _ => 66'd0;
        }
        no_fire(out.p.0, din[0].p.0, din[1].p.0, select.p.0) {}
    )
}
```

The above examples demonstrate some of the uses of the four *foreach* expressions. More examples include the relation, distributor, and flipflop. In the future, the use of *foreach* may be extended in order to create more sophisticated expansions. Also, more comparison operators may be introduced to control expansions.

4.13 *Type Resolution*

As shown in many of the previous examples, types can be specified as constants, as expressions in terms of other ports' types and constants, or left unspecified. Allowing such freedom provides the designer a means to design type polymorphic actors. In order to resolve the unspecified types, a type resolution algorithm was implemented. The type resolution algorithm propagates types form sources to sinks by using the default typing rules as well as the user defined types. Type resolution errors will be indicated in cases where there is a type conflict and where a type cannot be resolved. The type expressions have been developed in a way that type resolution proceeds in

a monotonic increasing manner. If it turns out that the types are not converging then a type resolution error indicating that a type resolution cycle has occurred is displayed.

When generating the operations, there can be unresolved types, but all types must be resolved before generating the simulators and hardware. As seen in previous examples, it is possible to resolve a type to a 0 width bit type. A 0 width type indicates that the port contains no information for the data path. It does not mean that the signal disappears. It is up to the controller synthesis to determine the true width of 0 width type ports.

4.14 *Methodology of Designing an Actor*

An actor is designed by describing the valid firing constraints and what actions should occur when the constraints are met. If possible an actor should be designed as a polymorphic actor. A designer must consider how his or her actor should behave in the context of a network of actors. After constructing a network of actors, the Tipi tools statically analyze and solve the constraints to determine all the valid source to sink minimal operations (independent operations are not combined to create product operations). Sources are all the input ports to a MescalPE and all the read ports of memory elements. Sinks are all the output ports of a MescalPE and all the write ports of memory elements. The set of minimal operations represent the architecture. The supporting control for these minimal operations are automatically generated. To summarize, the constraints specified by all the actors in a network are statically analyzed and solved in order to export the architecture as a set of minimal source to sink operations.

4.15 *Possible Future Enhancements*

There are two possible future enhancements that should be explored. The first extension would allow for a richer set of constraints. The second extension would allow actors to defer their definitions to black boxed components.

4.15.1 Quantifier Free First Order Logic on Constraints

As mentioned earlier, it would be nice to be able to specify constraints on control signals. There is no reason why we cannot support this in the existing framework. If one wants to always fire two memory actors at the same time, there is no way to use the existing hierarchy to make this happen. There are two ways that one could get this behavior. The first method is to combine the two memory actors into one actor. The second method is to use the constraints view to create a spatial constraint. Basically, we want a way to specify the constraints that would be currently specified in the constraints view in the actor language. This requires constraints across the hierarchy. Furthermore, to ease the specifying of constraints it would be nice to support quantifier free first order logic instead of only product of sums.

4.15.2 Read and Write Black Boxed Components

Currently simulation functions are read only. This certainly gives guarantees on the safety of the generated simulators, but fails to allow a designer to use a black boxed component within an actor. In the future, the language may support read and write black boxed components. The main disadvantage of supporting this feature is that it allows arbitrary code, which cannot be analyzed, to interact with the simulator.

5 Analyzable Actor Language Grammar

```
/* C-style /* */ and C++-style // comments are supported */

ID      ::= ((_*[a-zA-Z]+)|(_+[a-zA-Z0-9]+))[_a-zA-Z0-9]*
NUMBER ::= [0-9]+
OCTAL_NUMBER ::= [0-7]+
HEX_NUMBER ::= [0-9a-fA-F]+
BINARY_NUMBER ::= [0-1]+
SIMEXPR ::= [^#]*

precedence left ?, :
precedence left ||
precedence left &&
precedence left |, ~|
precedence left &, ~&, ^, ~^
precedence left ==, !=, <, <=, <, >=
precedence left <<, >>
precedence left +, -
precedence left *, /, %
precedence left !, UAND, UNAND, UOR, UNOR, UXNOR1, UXNOR2, ~
precedence left foreach, #begin sim, and, or, ID

actor               ::= ID ( port_list ) { program }

port_list           ::= ID , port_list
                        |
                        ID

program             ::= io_list memory_list rule_constraint
                        |
                        io_list rule_constraint

memory_list         ::= memory memory_list
                        |
                        memory

io_list             ::= io_list input
                        |
                        io_list output
                        |
                        input
                        |
                        output

memory              ::= reg ID ;
                        |
                        reg < type > ID ;
                        |
                        reg ID NUMBER ;
                        |
                        reg < type > ID NUMBER ;
                        |
                        ff ID ;
                        |
                        ff < type > ID ;
                        |
                        ff ID NUMBER ;
                        |
                        ff < type > ID NUMBER
```

```
input                 ::=  input io_definition ;

output                ::=  output io_definition ;

io_defintion          ::=  ID
                           |
                           < type > ID
                           |
                           @ ID
                           |
                           < type > @ ID

type                  ::=  max_type_expr
                           |
                           max_type_expr + type
                           |
                           ( max_type_expr )
                           |
                           max_type_expr - type_expr
                           |
                           type_expr

max_type_expr         ::=  max ( max_type_expr_list )

max_type_expr_list    ::=  type_expr , max_type_expr_list
                           |
                           type_expr

type_expr             ::=  type_expr + type_expr
                           |
                           type_expr - type_expr
                           |
                           log ( type_expr )
                           |
                           ceil ( type_expr )
                           |
                           floor ( type_expr )
                           |
                           ( type_expr )
                           |
                           type_id

type_id               ::=  NUMBER
                           |
                           ID . type
                           |
                           @ ID . type
                           |
                           instruction . type

/* currently, support only exists for POS formulations       */
/* arbitrary AND-OR expressions will eventually be supported */

rule_constraint       ::=  and ( rule_constraint_list )
                           |
                           or ( rule_constraint_list )
                           |
                           rule_list
                           |
                           foreach ( ID ) { rule_constraint_list }
```

```
rule_constraint_list  ::=  rule_constraint rule_constraint_list
                           |
                           rule_constraint

rule_list             ::=  rule rule_list
                           |
                           rule

rule                  ::=  ID ( symbol_decl_list ) {}
                           |
                           ID ( symbol_decl_list ) {statement_list}

symbol_decl           ::=  ID dotted
                           |
                           @ ID dotted
                           |
                           foreach_symbol_decl

symbol_decl_list      ::=  foreach_symbol , symbol_decl_list
                           |
                           symbol_decl , symbol_decl_list
                           |
                           foreach_symbol
                           |
                           symbol_decl

foreach_symbol        ::=  foreach ( ID ) {   foreach_cond
                                              foreach_symbol_decl }
                           |
                           foreach ( ID ) { foreach_symbol_decl }

foreach_symbol_decl   ::=  ID [ $ ID ] dotted

foreach_cond          ::=  if ( $ ID != $ ID )

statement_list        ::=  statement statement_list
                           |
                           statement

statement             ::=  let decl_list in statement_expr_list end
                           |
                           statement_expr_list

decl_list             ::=  decl decl_list
                           |
                           decl

decl                  ::=  val ID = expr ;
                           |
                           val < type > ID = expr ;

statement_expr_list   ::=  statement_expr statement_expr_list
                           |
                           statement_expr

statement_expr        ::=  foreach ( ID ) { statement_expr }
                           |
                           #begin sim simulation_function #end
                           |
                           symbol_expr = expr ;
                           |
                           ID [ expr ] = expr ;
```

```
simulation_function   ::=  ID ( )
                           |
                           ID ( function_param_list )
                           |
                           ID ( ) SIMEXPR
                           |
                           ID ( function_param_list ) SIMEXPR

function_param_list   ::=  symbol_expr , function_param_list
                           |
                           symbol_expr

expr                  ::=  case_expr
                           |
                           ite_expr
                           |
                           symbol_expr
                           |
                           indexed_expr
                           |
                           concatenation_expr
                           |
                           replication_expr
                           |
                           const_expr
                           |
                           ( expr )
                           |
                           bool_expr ? expr : expr
                           |
                           expr + expr
                           |
                           expr - expr
                           |
                           expr * expr
                           |
                           ID [ expr ]
                           |
                           expr / expr
                           |
                           expr % expr
                           |
                           expr & expr
                           |
                           expr ~& expr
                           |
                           expr | expr
                           |
                           expr ~| expr
                           |
                           expr ^ expr
                           |
                           expr ~^ expr
                           |
                           expr ^~ expr
                           |
                           expr << expr
                           |
                           expr >> expr
                           |
                           ~ expr
                           |
```

```
                    & expr          UAND
                    |
                    ~& expr         UNAND
                    |
                    | expr          UOR
                    |
                    ~| expr         UNOR
                    |
                    ^ expr          UXOR
                    |
                    ~^ expr         UXNOR1
                    |
                    ^~ expr         UXNOR2

const_expr          ::= decimal_number
                    |
                    octal_number
                    |
                    hex_number
                    |
                    binary_number

decimal_number      ::= NUMBER
                    |
                    NUMBER decimal_base NUMBER
                    |
                    < type > decimal_base NUMBER

octal_number        ::= NUMBER octal_base NUMBER
                    |
                    < type > octal_base OCTAL_NUMBER

hex_number          ::= NUMBER hex_base NUMBER
                    |
                    < type > hex_base HEX_NUMBER

binary_number       ::= NUMBER binary_base NUMBER
                    |
                    < type > binary_base BINARY_NUMBER

decimal_base        ::= 'd | 'D

octal_base          ::= 'o | 'O

hex_base            ::= 'h | 'H

binary_base         ::= 'b | 'B
```

```
bool_expr           ::= expr == expr
                        |
                        expr != expr
                        |
                        expr < expr
                        |
                        expr <= expr
                        |
                        expr > expr
                        |
                        expr >= expr
                        |
                        ( bool_expr )
                        |
                        bool_expr && bool_expr
                        |
                        bool_expr || bool_expr
                        |
                        !bool_expr

case_expr           ::= case expr of case_statement

case_statement      ::= foreach ( ID ) { $ ID => expr | } _ => expr
                        |
                        match_expr_list _ => expr

match_expr_list     ::= match_expr match_expr_list
                        |
                        match_expr

match_expr          ::= NUMBER => expr |

ite_expr            ::= if bool_expr then expr else expr

concatenation_expr  ::= { concatenation_list }

concatenation_list  ::= expr , concatenation_list
                        |
                        expr

replication_expr    ::= NUMBER { expr }

dotted              ::= .p.1 | .p.0 | .p.1.e | .p.0.e
                        |
                        .p.1.op | .p.0.op

indexed_expr        ::= ( expr ) [ type ]
                        |
                        ( expr ) [ type : type ]

symbol_expr         ::= ID
                        |
                        ID [ type ]
                        |
                        ID [ type : type ]
                        |
                        ID [ $ ID ]
                        |
                        ID [ $ ID ] [ type ]
                        |
                        ID [ $ ID ] [ type : type ]
```

6 Probe Language

Probe actors have the same semantics as analyzable actors, but only support write expressions in the assignments section. *Probe* actors also have the requirements that their firing rule constraints are a tautology. The tautology requirement guarantees that *probe* actors do not introduce any new semantics to the design. Furthermore, *probe* actors can only have inputs.

6.1 *Write Expressions*

A write expression is used to print an expression to standard out. The format of a write expression is shown below.

```
__write(expr);
```

Any expression, as defined in the analyzable actor language, can be used as the argument to *__write*. An example *probe* actor is shown below:

```
Probe(din) {

    input din;

    or(
        fire(din.p.1) {
            __write(din);
        }
        no_fire(din.p.0) {}
    )
}
```

The *probe* actor is used to analyze the input signal. Note that the constraints are a tautology (i.e. if *din* is present do *fire*, else if *din* is not present do *no_fire*).

7 Probe Language Grammar

```
/* C-style /* */ and C++-style // comments are supported */

ID      ::= ((_*[a-zA-Z]+)|(_+[a-zA-Z0-9]+))[_a-zA-Z0-9]*
NUMBER ::= [0-9]+
OCTAL_NUMBER ::= [0-7]+
HEX_NUMBER ::= [0-9a-fA-F]+
BINARY_NUMBER ::= [0-1]+

precedence left ?, :
precedence left ||
precedence left &&
precedence left |, ~|
precedence left &, ~&, ^, ~^
precedence left ==, !=, <, <=, <, >=
precedence left <<, >>
precedence left +, -
precedence left *, /, %
precedence left !, UAND, UNAND, UOR, UNOR, UXNOR1, UXNOR2, ~
precedence left foreach, #begin sim, and, or, ID

actor               ::= ID ( port_list ) { program }

port_list           ::= ID , port_list
                        |
                        ID

program             ::= io_list rule_constraint

io_list             ::= io_list input
                        |
                        input

input               ::= input io_definition ;

io_defintion        ::= ID
                        |
                        < type > ID
                        |
                        @ ID
                        |
                        < type > @ ID

type                ::= max_type_expr
                        |
                        max_type_expr + type
                        |
                        ( max_type_expr )
                        |
                        max_type_expr - type_expr
                        |
                        type_expr

max_type_expr       ::= max ( max_type_expr_list )

max_type_expr_list  ::= type_expr , max_type_expr_list
                        |
                        type_expr
```

```
type_expr           ::=  type_expr + type_expr
                         |
                         type_expr - type_expr
                         |
                         log ( type_expr )
                         |
                         ceil ( type_expr )
                         |
                         floor ( type_expr )
                         |
                         ( type_expr )
                         |
                         type_id

type_id             ::=  NUMBER
                         |
                         ID . type
                         |
                         @ ID . type
                         |
                         instruction . type

/* currently, support only exists for POS formulations       */
/* arbitrary AND-OR expressions will eventually be supported */

rule_constraint     ::=  and ( rule_constraint_list )
                         |
                         or ( rule_constraint_list )
                         |
                         rule_list
                         |
                         foreach ( ID ) { rule_constraint_list }

rule_constraint_list ::= rule_constraint rule_constraint_list
                         |
                         rule_constraint

rule_list           ::=  rule rule_list
                         |
                         rule

rule                ::=  ID ( symbol_decl_list ) {}
                         |
                         ID ( symbol_decl_list ) {statement_list}

symbol_decl         ::=  ID dotted
                         |
                         @ ID dotted
                         |
                         foreach_symbol_decl

symbol_decl_list    ::=  foreach_symbol , symbol_decl_list
                         |
                         symbol_decl , symbol_decl_list
                         |
                         foreach_symbol
                         |
                         symbol_decl

foreach_symbol      ::=  foreach ( ID ) {   foreach_cond
                                            foreach_symbol_decl }
                         |
                         foreach ( ID ) { foreach_symbol_decl }
```

```
foreach_symbol_decl   ::=  ID [ $ ID ] dotted

foreach_cond          ::=  if ( $ ID != $ ID )

statement_list        ::=  statement statement_list
                           |
                           statement

statement             ::=  let decl_list in statement_expr_list end
                           |
                           statement_expr_list

decl_list             ::=  decl decl_list
                           |
                           decl

decl                  ::=  val ID = expr ;
                           |
                           val < type > ID = expr ;

statement_expr_list   ::=  statement_expr statement_expr_list
                           |
                           statement_expr

statement_expr        ::=  foreach ( ID ) { statement_expr }
                           |
                           __write ( expr ) ;

expr                  ::=  case_expr
                           |
                           ite_expr
                           |
                           symbol_expr
                           |
                           indexed_expr
                           |
                           concatenation_expr
                           |
                           replication_expr
                           |
                           const_expr
                           |
                           ( expr )
                           |
                           bool_expr ? expr : expr
                           |
                           expr + expr
                           |
                           expr - expr
                           |
                           expr * expr
                           |
                           expr / expr
                           |
                           expr % expr
                           |
                           expr & expr
                           |
                           expr ~& expr
                           |
                           expr | expr
                           |
```

```
                    expr ~| expr
                    |
                    expr ^ expr
                    |
                    expr ~^ expr
                    |
                    expr ^~ expr
                    |
                    expr << expr
                    |
                    expr >> expr
                    |
                    ~ expr
                    |
                    & expr          UAND
                    |
                    ~& expr         UNAND
                    |
                    | expr          UOR
                    |
                    ~| expr         UNOR
                    |
                    ^ expr          UXOR
                    |
                    ~^ expr         UXNOR1
                    |
                    ^~ expr         UXNOR2

const_expr          ::= decimal_number
                        |
                        octal_number
                        |
                        hex_number
                        |
                        binary_number

decimal_number      ::= NUMBER
                        |
                        NUMBER decimal_base NUMBER
                        |
                        < type > decimal_base NUMBER

octal_number        ::= NUMBER octal_base NUMBER
                        |
                        < type > octal_base OCTAL_NUMBER

hex_number          ::= NUMBER hex_base NUMBER
                        |
                        < type > hex_base HEX_NUMBER

binary_number       ::= NUMBER binary_base NUMBER
                        |
                        < type > binary_base BINARY_NUMBER

decimal_base        ::= 'd | 'D

octal_base          ::= 'o | 'O

hex_base            ::= 'h | 'H

binary_base         ::= 'b | 'B
```

```
bool_expr          ::=  expr == expr
                        |
                        expr != expr
                        |
                        expr < expr
                        |
                        expr <= expr
                        |
                        expr > expr
                        |
                        expr >= expr
                        |
                        ( bool_expr )
                        |
                        bool_expr && bool_expr
                        |
                        bool_expr || bool_expr
                        |
                        !bool_expr

case_expr          ::=  case expr of case_statement

case_statement     ::=  foreach ( ID ) { $ ID => expr | } _ => expr
                        |
                        match_expr_list _ => expr

match_expr_list    ::=  match_expr match_expr_list
                        |
                        match_expr

match_expr         ::=  NUMBER => expr |

ite_expr           ::=  if bool_expr then expr else expr

concatenation_expr ::=  { concatenation_list }

concatenation_list ::=  expr , concatenation_list
                        |
                        expr

replication_expr   ::=  NUMBER { expr }

dotted             ::=  .p.1 | .p.0 | .p.1.e | .p.0.e
                        |
                        .p.1.op | .p.0.op

indexed_expr       ::=  ( expr ) [ type ]
                        |
                        ( expr ) [ type : type ]

symbol_expr        ::=  ID
                        |
                        ID [ type ]
                        |
                        ID [ type : type ]
                        |
                        ID [ $ ID ]
                        |
                        ID [ $ ID ] [ type ]
                        |
                        ID [ $ ID ] [ type : type ]
```

8 Simulator-Only Language

Simulation-only actors have the same semantics as *probe* actors, but only support simulation functions in the assignments section. *Simulation-only* actors also have the requirements that their constraints are a tautology. The tautology requirement guarantees that simulation only actors do not introduce any new semantics to the design. Since simulation functions are read-only black boxed components, these actors are passive and cannot influence the simulation. Furthermore, these actors can only have inputs.

9 Simulator-Only Language Grammar

```
/* C-style /* */ and C++-style // comments are supported */

ID     ::= ((_*[a-zA-Z]+)|(_+[a-zA-Z0-9]+))[_a-zA-Z0-9]*
NUMBER ::= [0-9]+
SIMEXPR ::= [^#]*

precedence left foreach, and, or, ID

actor                ::= ID ( port_list ) { program }

port_list            ::= ID , port_list
                         |
                         ID

program              ::= io_list rule_constraint

io_list              ::= io_list input
                         |
                         input

input                ::= input io_definition ;

io_defintion         ::= ID
                         |
                         < type > ID
                         |
                         @ ID
                         |
                         < type > @ ID

type                 ::= max_type_expr
                         |
                         max_type_expr + type
                         |
                         ( max_type_expr )
                         |
                         max_type_expr - type_expr
                         |
                         type_expr

max_type_expr        ::= max ( max_type_expr_list )

max_type_expr_list   ::= type_expr , max_type_expr_list
                         |
                         type_expr

type_expr            ::= type_expr + type_expr
                         |
                         type_expr - type_expr
                         |
                         log ( type_expr )
                         |
                         ceil ( type_expr )
                         |
                         floor ( type_expr )
                         |
                         ( type_expr )
                         |
```

```
                        type_id

type_id          ::=    NUMBER
                        |
                        ID . type
                        |
                        @ ID . type
                        |
                        instruction . type

/* currently, support only exists for POS formulations       */
/* arbitrary AND-OR expressions will eventually be supported */

rule_constraint  ::=    and ( rule_constraint_list )
                        |
                        or ( rule_constraint_list )
                        |
                        rule_list
                        |
                        foreach ( ID ) { rule_constraint_list }

rule_constraint_list ::= rule_constraint rule_constraint_list
                        |
                        rule_constraint

rule_list        ::=    rule rule_list
                        |
                        rule

rule             ::=    ID ( symbol_decl_list ) {}
                        |
                        ID ( symbol_decl_list ) {statement_list}

symbol_decl      ::=    ID dotted
                        |
                        @ ID dotted
                        |
                        foreach_symbol_decl

symbol_decl_list ::=    foreach_symbol , symbol_decl_list
                        |
                        symbol_decl , symbol_decl_list
                        |
                        foreach_symbol
                        |
                        symbol_decl

foreach_symbol   ::=    foreach ( ID ) { foreach_cond
                                         foreach_symbol_decl }
                        |
                        foreach ( ID ) { foreach_symbol_decl }

foreach_symbol_decl ::= ID [ $ ID ] dotted

foreach_cond     ::=    if ( $ ID != $ ID )

statement_list   ::=    statement statement_list
                        |
                        statement

statement        ::=    statement_expr_list

statement_expr_list ::= statement_expr statement_expr_list
```

```
                        |
                        statement_expr

statement_expr      ::= foreach ( ID ) { statement_expr }
                        |
                        simulation_function

simulation_function ::= ID ( )
                        |
                        ID ( function_param_list )
                        |
                        ID ( ) SIMEXPR
                        |
                        ID ( function_param_list ) SIMEXPR

function_param_list ::= symbol_expr , function_param_list
                        |
                        symbol_expr

dotted              ::= .p.1 | .p.0 | .p.1.e | .p.0.e
                        |
                        .p.1.op | .p.0.op

symbol_expr         ::= ID
                        |
                        ID [ type ]
                        |
                        ID [ type : type ]
                        |
                        ID [ $ ID ]
                        |
                        ID [ $ ID ] [ type ]
                        |
                        ID [ $ ID ] [ type : type ]
```

10 Type System

The type system supports two types. The first type is an arbitrary width unsigned integer (two's compliment). The second type is Boolean. Inputs and outputs of actors must be unsigned integers, but within the expressions for an actor, Booleans can be used.

10.1 *Type Resolution Rules*

```
t1 : UINT(n) t2 : UINT(m)
-------------------------------------------------
t1 (+,-,*,%,&,~&,|,~|,^,~^,^~) t2 : UINT(max(n, m))


t1 : UINT(n) t2 : UINT(m) t3 : UINT(m) ... tN+1 : UINT(m)
-------------------------------------------------------
case t1 of n1 => t2 | n2 => t3 | _ tN+1 : UINT(m)


t1 : UINT(n1) t2 : UINT(n2) ... tN : UINT(N)
-------------------------------------------
(t1, t2, … tN) : UINT(n1 + n2 + … N)


t1 : bool t2 : UINT(n) t3 : UINT(m)
-----------------------------------
t1 ? t2 : t3 : UINT(max(n, m))


t1 : bool t2 : UINT(n) t3 : UINT(m)
-----------------------------------
if t1 then t2 else t3 : UINT(max(n, m))


t1 : UINT(n) t2 : UINT(m)
-------------------------
t1 / t2 : UINT(n)


t1 : UINT(n)
----------------------------------------
(t1)[msb:lsb] : UINT(abs(msb - lsb) + 1)


t1 : UINT(n)
-----------------
n1 => t1 : UINT(n)


t : UINT(n)
-------------------
N {t} : UINT(N * n)


t1 : UINT(n) t2 : UINT(m)
-------------------------
t1 (<<,>>) t2 : UINT(n)
```

```
t : UINT(n)
-------------------------
mem[t1] : UINT(width_mem)

t : UINT(n)
---------------
~t : UINT(n)

t : UINT(n)
-----------------------------
(&,~&,|,~|,^,~^,^~)t : UINT(1)

t1 : UINT(n) t2 : UINT(m)
-----------------------------
t1 (==,!=,<,<=,>,>=) t2 : bool

t1 : bool t2 : bool
--------------------
t1 (&&,||) t2 : bool

t : bool
---------
!t : bool
```

11 Assembly Language

The assembly section allows for the description of a program in terms of operations. Each line of an assembly program represents a set of operations separated by commas that are executed in parallel (this is called an instruction). If an operation requires arguments, then they are specified after the operation. Consecutive lines represent a sequence of instructions to be executed. An example assembly program is shown below:

```
{i_1(), i_2(2)}
{i_3(), i_4(1,100)}
```

The above program executes *i_1()* and *i_2(2)* in parallel and then executes *i_3()* and *i_4(1,100)* in parallel. Also the lines can be thought of location 0 and 1. This allows for the implementation of jumps.

12 Assembly Language Grammar

```
/* C-style /* */ and C++-style // comments are supported */

ID     ::= ((_*[a-zA-Z]+)|(_+[a-zA-Z0-9]+))[_a-zA-Z0-9]*
NUMBER ::= [0-9]+

instructions              ::= labeled_instruction_set instructions
                              |
                              labeled_instruction_set

labeled_instruction_set   ::= ID : instruction_set
                              |
                              instruction_set

instruction_set           ::= { instruction_list }

instruction_list          ::= instruction , instruction_list
                              |
                              instruction

instruction               ::= ID ( )
                              |
                              ID ( parameter_list )

parameter_list            ::= parameter , parameter_list
                              |
                              parameter

parameter                 ::= NUMBER
                              |
                              ID
```

13 Constraints Language

The constraints language is utilized to specify spatial and temporal constraints. Constraints can be specified between operations, between operations and constraints, and between constraints. Spatial constraints are specified using ||. Temporal constraints are specified using ;. An example constraint is shown below:

```
foo(a,b,c) = (i_1(a) || i_2(b,c)) ; i_3(4)
```

The *foo(a,b,c)* constraint says that *i_1(a)* and *i_2(b,c)* should execute in parallel, followed by *i_3(4)*. Notice how variables and constants can be used.

14 Constraints Language Grammar

```
/* C-style /* */ and C++-style // comments are supported */

ID     ::= ((_*[a-zA-Z]+)|(_+[a-zA-Z0-9]+))[_a-zA-Z0-9]*
NUMBER ::= [0-9]+

precedence right ;
precedence right ||

constraints    ::= constraints ; constraints
                   |
                   constraints || constraints
                   |
                   ( constraints )
                   |
                   constraint

constraint     ::= ID
                   |
                   ID ( )
                   |
                   ID ( parameter_list )

parameter_list ::= parameter , parameter_list
                   |
                   parameter

parameter      ::= NUMBER
                   |
                   ID
```

15 Acknowledgement

The Mescal project and the Tipi framework are supported, in part, by Infineon Technologies and the Microelectronics Advanced Research Consortium (MARCO), and are part of the efforts of the Gigascale Systems Research Center (GSRC).