"System Verilog Tagged Unions and Pattern Matching"

(An extension to System Verilog 3.1 proposed to Accellera)

Bluespec, Inc.

Contact:
Rishiyur S. Nikhil, CTO, Bluespec, Inc.
200 West Street, 4th Flr., Waltham, MA 02541, USA
Email: nikhilbluespec.com Phone: +1 (781) 250 2203

Ver. 2, December 5, 2003
(see rationale in previous docs Sep 9 and Oct 3)

Introduction

We submitted a proposal for Tagged Unions and Pattern Matching on Sep 9 (revised Oct 3). It was discussed for the first time at the SV-BC meeting Nov 10, with extensive feedback. This is a revised proposal. It focuses on the proposal per se, omitting the extensive rationale in the previous document.

We have structured the proposal into 2 parts:

I. Tagged unions for type-safety and brevity.
Benefits: Type-safety and brevity. Type-safety improves correctness. Tagged unions also benefit formal verification because it improves the ability to reason about programs.

II. Pattern matching in **case** statements, **if** statements and conditional expressions.
Benefits: Dramatic improvement in brevity and readability. Makes it easier to implement tagged unions without runtime checks. Also improves formal reasoning.

(The last few pages contain a checklist of issues/suggestions raised on Nov 10, and subsequently by David Smith, Brad Pierce and Yong Xiao, together with a brief description of how each has been handled.)

These constructs significantly raise the level of programming with structures and unions, eliminate a number of common errors, and improve readability. The underlying ideas have had an enthusiastic following in many high-level languages for many years, in no small part because they have clean formal semantics. They are fully synthesizable (> 3 years experience with synthesis), and so are of interest both to designers and to verification engineers.

Part I: Tagged Unions

In the syntax box at the top of "Section 3.11 Structures and Unions" add the optional prefix **tagged** *to the* **union** *keyword, and extend struct_union_member to allow* **void** *as the type for tagged union members:*

```
                                                          // from Annex A.2.2.1
...
struct_union ::= struct | [ tagged ] union
...
struct_union_member ::=
      { attribute_instance } variable_declaration ;
   | { attribute_instance } void list_of_variable_identifiers ;        // in tagged unions only
```

At the end of Section 3.11, add the following text.

The keyword **tagged** preceding a union declares it as a tagged union, thereby making it into a type-checked union. An ordinary (untagged) union can be updated using a value of one member type and read as a value of another member type, which is a potential type loophole. A tagged union stores both the member value and a *tag*, i.e., additional bits representing the current member name. The tag and value can only be updated together consistently, using a statically type-checked tagged union expression (Section 7.14+). The member value can only be read with a type that is consistent with the current tag value (i.e., member name). Thus, it is impossible to store a value of one type and (mis)interpret the bits as another type.

In addition to type safety, the use of member names as tags also makes code simpler and smaller than code that has to track unions with explicit tags. Tagged unions can also be used with pattern matching (Section 8.4), which improves readability even further.

In tagged unions, members can be declared with type **void**, when all the information is in the tag itself, as in the following example of an integer together with a valid bit:

```
typedef tagged union {
    void  Invalid;
    int   Valid;
} VInt;
```

A value of `VInt` type is either Invalid and contains nothing, or is Valid and contains an int. Section 7.14+ describes how to construct values of this type, and also describes how it is impossible to read an integer out of a `VInt` value that currently has the `Invalid` tag.

Example:

```
typedef tagged union {
    struct {
        bit [4:0] reg1, reg2, regd;
    } Add;
    tagged union {
        bit [9:0] JmpU;
```

```
        struct {
             bit [1:0] cc,
             bit [9:0] addr;
        } JmpC;
    } Jmp;
} Instr;
```

A value of `Instr` type is either an Add instruction, in which case it contains three 5-bit register fields, or it is a Jump instruction. In the latter case, it is either an unconditional jump, in which case it contains a 10-bit destination address, or it is a conditional jump, in which case it contains a 2-bit condition-code register field and a 10-bit destination address. Section 7.14+ describes how to construct values of `Instr` type, and describes how, in order to read the `cc` field, for example, the instruction must have opcode `Jmp` and sub-opcode `JmpC`.

When the **packed** qualifier is used on a tagged union, all the members must have packed types, but they do not have to be of the same size. The (standard) representation for a packed tagged union is the following.

- The size is always equal to the number of bits needed to represent the tag plus the maximum of the sizes of the members.
- The size of the tag is the minimum number of bits needed to code for all the member names (e.g., 5 to 8 members would need 3 tag bits).
- The tag bits are always left-justified (i.e., towards the most-significant bits).
- For each member, the member bits are always right-justified (i.e., towards the least-significant bits).
- The bits between the tag bits and the member bits are undefined (shall contain '`x`'). In the extreme case of a `void` member, only the tag is significant and all the remaining bits are undefined.

The representation scheme is applied recursively to any nested tagged unions.

Example: If the `VInt` type definition had the **packed** qualifier, Invalid and Valid values will have the following layouts, respectively:

```
 1                               32
+-+-------------------------------------------------------------+
|0| x x x x x x x ...          ...        ...   x x x x x x x x x |
+-+-------------------------------------------------------------+

 1                               32
+-+-------------------------------------------------------------+
|1|                      ... an int value ...                   |
+-+-------------------------------------------------------------+
 ^
 +----- tag is 0 for Invalid, 1 for Valid
```

Example: If the `Instr` type had the **packed** qualifier, its values will have the following layouts:

```
 1     5         5         5
+-+---------+---------+---------+
|0|  reg1   |  reg2   |  regd   |           Add instrs
+-+---------+---------+---------+

 1  2  1  2         10
+-+---+-+---+-------------------+
|1| xx|0| xx|                   |           Jmp/JmpU instrs
+-+---+-+---+-------------------+

 1  2  1  2         10
+-+---+-+---+-------------------+
|1| xx|1| cc|       addr        |           Jmp/JmpC instrs
+-+---+-+---+-------------------+
 ^     ^
 |     +----- 0 for JmpU, 1 for JmpC
 |
 +----- 0 for Add, 1 for Jmp
```

Add the following new sub-section just after "Section 7.14 Structure expressions":

Section 7.14+ Tagged union expressions and member access

expression ::= *from Annex A.8.3*
  ...
  | tagged_union_expression

tagged_union_expression ::=
  tagged member_identifier [expression]

A tagged union expression (packed or unpacked) is expressed using the keyword **tagged** followed by a tagged union member identifier, followed by an expression representing the corresponding member value. For `void` members the member value expression is omitted.

Example:

```
typedef tagged union {
    void  Invalid;
    int   Valid;
} VInt;

VInt vi1, vi2;

vi1 = tagged Valid   (23+34);                    // Create Valid int
vi2 = tagged Invalid;                            // Create an Invalid value
```

In the tagged union expressions below, the expressions in braces are structure expressions (Section 7.14).

```
typedef tagged union {
    struct {
        bit [4:0] reg1, reg2, regd;
    } Add;
    tagged union {
        bit [9:0] JmpU;
        struct {
             bit [1:0] cc,
             bit [9:0] addr;
        } JmpC;
    } Jmp;
} Instr;

Instr i1, i2;

// Create an Add instruction with its 3 register fields
i1 = ( e
     ? tagged Add { e1, 4, ed };                  // struct members by position
     : tagged Add { reg2:e2, regd:3, reg1:19 }); // by name (order irrelevant)

// Create a Jump instruction, with ‘‘unconditional’’ sub-opcode
i1 = tagged Jmp (tagged JmpU 239);

// Create a Jump instruction, with ‘‘conditional’’ sub-opcode
i2 = tagged Jmp (tagged JmpC { 2, 83 });         // inner struct by position
i2 = tagged Jmp (tagged JmpC { cc:2, addr:83 }); // by name
```

The type of a tagged union expression must be known from its context (e.g., it is used in the RHS of an assignment to a variable whose type is known, or it is has a cast, or it is used inside another expression from which its type is known). The expression evaluates to a tagged union value of that type. The tagged union expression can be completely type-checked statically: the only member names allowed after the **tagged** keyword are the member names for the expression type, and the member expression must have the corresponding member type.

An uninitialized variable of tagged union type shall contain `x`'s in all the bits, including the tag bits. A variable of tagged union type can be initialized with a tagged union expression provided the member value expression is a legal initializer for the member type.

Members of tagged unions can be read or assigned using the usual dot-notation. Such accesses are completely type-checked, i.e., the value read or assigned must be consistent with the current tag. In general, this may require a runtime check. An attempt to read or assign a value whose type is inconsistent with the tag results in a runtime error.

All the following examples are legal only if the instruction variable `instr` currently has tag Add:

```
x           = i1.Add.reg1;
i1.Add      = {19, 4, 3};
i1.Add.reg2 = 4;
```

In "Annex A Formal Syntax" make all the BNF changes described in this part of the proposal.

In "Annex B Keywords", add the keyword **tagged**.

Part II: Pattern Matching in Case Statements, If Statements and Conditional Expressions

Change the syntax box at the start of "Section 7.16 Conditional operator":

conditional_expression ::= *from Annex A.8.3*
    cond_predicate **?** { attribute_instance } expression **:** expression

cond_predicate ::= *from Annex A.6.6*
    expression_or_cond_pattern { **&&** expression_or_cond_pattern }

expression_or_cond_pattern ::=
    expression | cond_pattern

cond_pattern ::= expression **matches** pattern

Insert the following after the syntax box and before the first sentence in "Section 7.16 Conditional operator":

> This section describes the traditional notation where cond_predicate is just a single expression. SystemVerilog also allows cond_predicate to perform pattern matching, and this is described in Section 8.4.

In the syntax box at the top of "Section 8.4 Selection Statements", change predicates in if-statements to allow cond_patterns, add productions for cond_patterns, and add a new family of clauses to the case_statement production for pattern matching case statements:

```
conditional_statement ::=                                    from Annex A.6.6
     if ( cond_predicate ) statement_or_null
   | unique_priority_if_statement

unique_priority_if_statement ::=
     [ unique_priority ] if ( cond_predicate ) statement_or_null
     { else if ( cond_predicate ) statement_or_null }
     [ else statement_or_null ]

unique_priority ::= unique | priority

cond_predicate ::=
     expression_or_cond_pattern { && expression_or_cond_pattern }

expression_or_cond_pattern ::=
     expression | cond_pattern

cond_pattern ::= expression matches pattern

case_statement ::=                                           // from Annex A.6.7
     [ unique_priority ] case_keyword ( expression ) case_item { case_item } endcase
   | [ unique_priority ] casep_keyword ( expression ) casep_item { casep_item } endcase

case_keyword ::= case | casez | casex
case_item ::=
     expression { , expression } : statement_or_null
   | default [ : ] statement_or_null

casep_keyword ::= casep | casepz | casepx
casep_item ::=
     pattern [ &&& expression ] : statement_or_null
   | default [ : ] statement_or_null
```

At the end of "Section 8.4 Selection Statements", add the following sub-section:

Section 8.4.1 Pattern matching

Pattern matching provides a visual and succinct notation to compare a value against structures, tagged unions and constants, and to access their members. SystemVerilog adds pattern matching capability to **case** and **if** statements, and to conditional expressions. Before describing pattern matching in those contexts, we first describe the general concepts.

A pattern is a nesting of tagged union and structure expressions with identifiers, constant expressions, and the wildcard pattern $ at the leaves. For tagged union patterns, the identifier following the **tagged** keyword is a union member name. For `void` members the nested member pattern is omitted.

```
pattern ::=                                                    // from Annex A.6.7.1
      pattern_identifier
    | $
    | . constant_expression
    | tagged member_identifier [ pattern ]
    | { pattern , ... , pattern }
    | { member_identifier : pattern , ... , member_identifier : pattern }

pattern_identifier ::= identifier
```

A pattern always occurs in a context of known type because it is matched against an expression of known type. Recursively, its nested patterns also have known type. A constant expression pattern must be of integral type. Thus a pattern can always be statically type-checked.

Each pattern introduces a new scope; the extent of this scope is described separately below for **case** statements, **if** statements and conditional expressions. Each pattern identifier is implicitly declared as a new variable within the pattern's scope, with the unique type that it must have based on its position in the pattern. Pattern identifiers must be unique in the pattern, i.e., the same identifier cannot be used in more than one position in a single pattern.

We always match the value V of an expression against a pattern. Note that static type-checking ensures that V and the pattern have the same type. The result of a pattern match is:

- A 1-bit determined value: 0 (false, or fail) or 1 (true, or succeed). The result cannot be `x` or `z` even if the value and pattern contain such bits.
- If the match succeeds, the pattern identifiers are bound to the corresponding members from V, using ordinary procedural assignment.

Each pattern is matched using the following simple recursive rule:

- An identifier pattern always succeeds (matches any value), and the identifier is bound to that value (using ordinary procedural assignment).
- The wildcard pattern $ always succeeds.
- A constant expression pattern succeeds if V is equal to its value.
- A tagged union pattern succeeds if the value has the same tag and, recursively, if the nested pattern matches the member value of the tagged union.
- A structure pattern succeeds if, recursively, each of the nested member patterns matches the corresponding member values in V. In structure patterns with named members, the textual order of members does not matter, and members may be omitted. Omitted members are ignored.

Conceptually, if the value V is seen as a flattened vector of bits, the pattern specifies which bits to match, with what values they should be matched and, if the match is successful, which bits to extract and bind to the pattern identifiers. Matching can be insenstitive to `x` and `z` values, as described in the individual constructs below.

Section 8.4.1.1 Pattern matching in case statements

A **casep** statement consists of an expression in parentheses followed by a series of "casep_items". The left-hand side of a case item, before the ":", consists of a pattern and, optionallly, the

operator && followed by a boolean "filter" expression. The right-hand side of a case item is a statement. Each pattern introduces a new scope, in which its pattern identifiers are implicitly declared; this scope extends to optional filter expression and the statement in the right-hand side of the same case item. Thus different case items can reuse pattern identifiers.

All the patterns are completely statically type-checked. The expression being tested in the **casep** statement must have a known type, which is the same as the type of the pattern in each case item.

The expression in parentheses in a **casep** statement shall be evaluated exactly once. Its value V shall be matched against the left-hand sides of the case items, one at a time, in the exact order given, ignoring the default case item if any. During this linear search, if a case item is selected, its statement is executed and the linear search is terminated. If no case item is selected, and a default case item is given, then its statement is executed. If no case item is selected and no default case item is given, no statement is executed.

For a case item to be selected, the value V must match the pattern (and the pattern identifiers are assigned the corresponding member values in V) and then the boolean filter expression must evaluate to true (a determined value other than 0).

Example:

```
typedef tagged union {
    void  Invalid;
    int   Valid;
} VInt;
...
VInt v;
...
casep (v)
  tagged Invalid : $display ("v is invalid");
  tagged Valid  n: $display ("v is Valid with value %d", n);
endcase
```

In the case statement, if `v` currently has the Invalid tag, the first pattern is matched. Otherwise, it must have the Valid tag, and the second pattern is matched. The identifier `n` is bound to the value of the Valid member, and this value is displayed. It is impossible to access the integer member value (`n`) when the tag is Invalid.

Example:

```
typedef tagged union {
    struct {
        bit [4:0] reg1, reg2, regd;
    } Add;
    tagged union {
        bit [9:0] JmpU;
        struct {
             bit [1:0] cc,
             bit [9:0] addr;
        } JmpC;
```

```
    } Jmp;
} Instr;
...
Instr instr;
...
casep (instr)
  tagged Add {r1,r2,rd} && (rd != 0): rf[rd] = rf[r1] + rf[r2];
  tagged Jmp j                      : casep (j)
                                        tagged JmpU  a   : pc = pc + a;
                                        tagged JmpC {c,a}: if (rf[c]) pc = a;
                                      endcase
endcase
```

If `instr` holds an Add instruction, the first pattern is matched, and the identifiers `r1`, `r2` and `rd` are bound to the three register fields in the nested structure value. The right-hand side statement executes the instruction on the register file `rf`. It is impossible to access these register fields if the tag is Jmp. If `instr` holds a Jmp instruction, the second pattern is matched, and the identifier `j` is bound to the nested tagged union value. The inner **casep** statement, in turn, matches this value against JmpU and JmpC patterns, and so on.

Example (same as previous example, but using wildcard and constant patterns to eliminate the `rd` = 0 case:

```
casep (instr)
  tagged Add { $, $, . 0}: ;    // no op
  tagged Add {r1,r2,  rd}: rf[rd] = rf[r1] + rf[r2];
  tagged Jmp j           : casep (j)
                             tagged JmpU  a   : pc = pc + a;
                             tagged JmpC {c,a}: if (rf[c]) pc = a;
                           endcase
endcase
```

Example (same as previous example, but using nested patterns):

```
casep (instr)
  tagged Add {r1,r2,rd} && (rd != 0): rf[rd] = rf[r1] + rf[r2];
  tagged Jmp (tagged JmpU a)        : pc = pc + a;
  tagged Jmp (tagged JmpC {c,a})    : if (rf[c]) pc = a;
endcase
```

Example (same as previous example, with named structure components):

```
casep (instr)
  tagged Add {reg2:r2,regd:rd,reg1:r1} && (rd != 0): rf[rd] = rf[r1] + rf[r2];
  tagged Jmp (tagged JmpU a)                       : pc = pc + a;
  tagged Jmp (tagged JmpC {addr:a,cc:c})           : if (rf[c]) pc = a;
endcase
```

casep, **casepz** and **casepx** are analogous to **case**, **casez** and **casex**, respectively. In other words, during pattern matching, wherever two bits are compared (whether they are tag bits or members), the **casez** form ignores `z` bits, and the **casex** form ignores both `z` and `x` bits.

The **priority** and **unique** qualifiers play their usual role. **priority** implies that some case item must be selected. **unique** also implies that exactly one case item will be selected, so that they may be evaluated in parallel.

Section 8.4.1.2 Pattern matching in if statements

The predicate in an **if** statement can be a series of clauses separated with the && operator. Each clause is either an expression (used as a boolean), or has the form *expression* **matches** *pattern*. The clauses represent a sequential conjunction from left to right, i.e., if any clause fails the remaining clauses are not evaluated, and all of them must succeed for the predicate to be true. Boolean expression clauses are evaluated as usual. Each pattern introduces a new scope, in which its pattern identifiers are implicitly declared; this scope extends to the remaining clauses in the predicate and to the corresponding "true" arm of the **if** statement.

In each e **matches** p clause, e and p must have the same known statically known type. The value of e is matched against the pattern p as described above.

Even though pattern matching clauses always return a defined 1-bit result, they may be conjoined with ordinary expressions in the predicate, and hence the overall result may be ambiguous. The standard semantics of **if** statements hold, i.e., the first statement is executed if and only if the result is a determined value other than 0.

Example:

```
if (e matches (tagged Jmp (tagged JmpC {cc:c,addr:a})))
     ...     // c and a can be used here
else
     ...
```

Example (same as previous example, illustrating a sequence of two pattern-matches with identifiers bound in the first pattern used in the second pattern).

```
if (e matches (tagged Jmp j),
    j matches (tagged JmpC {cc:c,addr:a}))
    ...     // c and a can be used here
else
    ...
```

Example (same as first example, but adding a boolean expression to the sequence of clauses). The idea expressed is: "if e is a conditional jump instruction and the condition register is not equal to zero ...".

```
if (e matches (tagged Jmp (tagged JmpC {cc:c,addr:a}))
    && (rf[c] != 0))
     ...     // c and a can be used here
else
     ...
```

The **priority** and **unique** qualifiers play their usual role for **if** statements even if they use pattern matching.

Section 8.4.1.3 Pattern matching in conditional expressions

A conditional expression *(e1 ? e2 : e3)* can also use pattern matching, i.e., the predicate $e1$ can be a sequence of expressions and "expression **matches** pattern" clauses separated with the && operator, just like the predicate of an **if** statement. The clauses represent a sequential conjunction from left to right, i.e., if any clause fails the remaining clauses are not evaluated, and all of them must succeed for the predicate to be true. Boolean expression clauses are evaluated as usual. Each pattern introduces a new scope, in which its pattern identifiers are implicitly declared; this scope extends to the remaining clauses in the predicate and to the consequent expression *e2*.

As described in the prevous section, $e1$ may evaluate to true, false or an ambiguous value. The semantics of the overall conditional expression are described in Section 7.16, based on these three possible outcomes for $e1$.

In "Annex A Formal Syntax" make all the BNF changes described in this part of the proposal.

In "Annex B Keywords" add the keywords **casep**, **casepz**, **casepx** *and* **matches**.

Checklist of issues raised in the Nov 10 meeting in San Jose, plus issues raised by Brad Pierce, Yong Xiao and David Smith in e-mail

The following is a checklist of issues raised the first time the "Tagged Unions and Pattern Matching" proposal was presented before the SV-BC committee, on Nov 10, 2003 at San Jose (for which, many thanks!), plus more issues raised by Brad, Yong and David Smith on a revised proposal of Nov 19. For each issue, I have stated the issue, and described briefly how it has been addressed.

Implicitly declared identifiers and their scope

The issue: The implicit declaration of pattern identifiers does not follow Verilog tradition, which does not implicitly declare identifiers.

Response: Yes, but it is worth it, for the following reasons:

- The trend in modern languages is to allow this, for simplicity, brevity and readability, provided the type and scope of the identifier is unambiguous and simply defined.
- In the pattern matching proposal, the type of the identifier is obvious and unique, based on its position in the pattern, and the scope is simple and very local: the right-hand side of a case item; the true-consequent in an if-statement; the true-consequent in a conditional expression.
- The **foreach** proposal (Section 12.4.7) also introduces implicitly declared identifiers.

Distinguishing Pattern Identifiers from Constants

The issue: for an identifier in a pattern, how do we know whether it represents a constant currently in scope (such an enum identifier or a parameter) to be checked for equality with the tested value, or whether it represents a new, implicitly declared pattern identifier to be bound to the value?

Response: The syntax distinguishes these situations. A pattern identifier appears directly, by itself. A constant identifier can only be part of a constant expression in the pattern, and these are always preceded by the "`.`" symbol. So, there is no ambiguity. (Yong Xiao suggested this syntax.)

Can Pattern Identifiers be used for update?

The issue: What exactly are pattern identifiers bound to? A copy of the *value* of the corresponding member? Can these identifiers be assigned? If so, does the original member change? I.e., is the identifier bound to an l-value or an r-value representing the member?

Response The text has been clarified to explain that pattern matching is only used to access member values, not for assignment. Pattern identifiers represent fresh variables, and on a successful pattern-match they receive a copy of the corresponding member values using ordinary procedural assignment. Yes, these identifiers can be assigned, but that has no effect on any other structure or tagged union. The reason for these semantics is that pattern matching can test an arbitrary expression of the appropriate type, and so it does not make sense to think of the pattern identifiers as updatable l-values. To update an original structure or tagged union, one uses the standard update notations. Also, experience shows that one rarely updates individual members; if the structure or tagged union is ever updated, one usually assigns it as a whole item using a tagged union expression.

Ambiguity between pattern matching and normal case semantics

The issue: Since the left-hand side of a case item can be an arbitary runtime expression, and since patterns are a subset of expressions, how do we know if a case statement has normal semantics or pattern matching semantics?

Response: Correct: it would be impossible to disambiguate. We have introduced a new kind of case statement using the keywords **casep**, **casepx** and **casepz** which are used exclusively for pattern matching.

Subsequent issue: David Smith asked if we could use the syntax **wildcard case/casex/casez** instead of introducing these new keywords, since **wildcard** is a new keyword in the SV-EC functional coverage proposal, and seems related.

Response: I have read the SV-EC functional coverage proposal and I don't see **wildcard** as having a natural usage here with pattern matching. Here are two alternatives that may work:

- Introduce a single new keyword **pattern**, so that the construct look like:
  pattern case/casex/casez (...) ... casep_items ... **endcase**
- Re-use the keyword **matches** just after the expression being tested, so that the construct would look like:
  case/casex/casez (expression) **matches** ... casep_items ... **endcase**

What is the SV-BC committee's consensus?

Behavior of x and z values in pattern matching

The issue: What are the semantics `x` and `z` values in **case**, **if** and conditional expressions, in the presence of pattern matching?

Response: These have been precisely defined in this proposal. There are no surprises: the behavior follows the normal behavior of these constructs without pattern matching.

Behavior of unique and priority keywords in pattern matching case and if statements

The issue: What are the semantics of the **unique** and **priority** qualifiers for **if** and **case**, in the presence of pattern matching?

Response: These have been precisely defined in this proposal. There are no surprises: the behavior follows the normal behavior of these constructs without pattern matching.

Semantics of pattern matching in if stmts (are pattern-matches really Boolean expressions)?

The issue: In the original proposal, "e1 **matches** pattern" was described as an ordinary Boolean expression. That raises tricky issues such as this: suppose we had:

```
if ((e1 matches p1) || (e2 matches p2)) ...
```

then what if p1 fails and p2 succeeds? What do the pattern variables in p1 (failed) represent? Can pattern variables in p1 (p2) be used in e2 (e1)? And similar questions arise in other Boolean contexts.

Response: "e1 **matches** pattern" is no longer an ordinary expression. The BNF has been fixed so that it can only be used at the top-level of the predicate in an if-statement predicate or conditional expression. The semantics of the phrase, and scope of pattern variables has been defined precisely and unambiguosly. The BNF is a proper extension, i.e., the existing BNF is a proper subset.

Can pattern matching be used in conditional expressions "(e1?e2:e3)"?

The issue: The original proposal only talked about pattern matching in **if** statements. What about conditional expressions?

Response: The proposal has been extended to allow pattern matching in conditional expressions as well. The new BNF is a proper extension, i.e., the existing BNF is a proper subset.

Simplifying BNF, clarifying language on defaults

The issue: Brad Pierce suggested numerous simplifications to the BNF based on the latest BNF proposals. Brad also suggested clarifying the language around defaults in casep statements.

Response: All these suggestions have been incorporated.

Don't care patterns

The issue: Yong Xiao asked if we could have a notation for "don't care" (wildcard) patterns, so that one did not have to invent a pattern identifier (that would not be used in the RHS of the case item).

Response: I have extended the syntax of patterns so that `$` can be used as a wildcard pattern.

Multiple alternatives in a casep item

The issue: Yong Xiao asked if it would make sense to have multiple patterns in a single casep item, just like we have multiple expressions in the LHS of a normal case item.

Response: I don't think this will work. In a normal case item, multiple expressions represent a disjunction (i.e., any one can match, to select the case item). Disjunctions don't make sense with patterns, since an unmatched pattern leaves the pattern identifiers undefined, and these identifiers may be used in the RHS.

Use of void type for tagged union members

The issue: David Smith pointed out that this is a new use of **void**, which previously was only allowed for function return types, and for casting a function call into a procedural statement. His suggestion: use some other type, or explain that this is a new usage.

Response: **void** is the natural thing to use in this situation. I have fixed up the BNF and the text to make this new usage clear.

Pack/Unpack part of the proposal has been removed

The issue: One part of the original proposal had to do with custom representations, involving a user-defined pair of functions called "pack/unpack" to convert to and from bits. David Smith suggested that this may overlap with the bit-stream proposal of SV-EC.

Response: Yes, the bit-stream proposal subsumes it. This part of the proposal has been removed.

Other style suggestions from David Smith

The issue: David made several suggestions to clarify the explanation (repeat examples instead of referring across sections; better examples on usage; better explanation of scope of pattern identifiers, some typos, etc.)

Response: All suggestions have been adopted.

Copyright notice

The issue: David Smith expressed a concern about the "(c) 2003 Bluespec, Inc." notice in the original proposal.

Response: The copyright notice has been removed.