



US 20060277534A1

(19) **United States**

(12) **Patent Application Publication**

Kasuya

(10) **Pub. No.: US 2006/0277534 A1**

(43) **Pub. Date: Dec. 7, 2006**

(54) **EVALUATION OF A TEMPORAL
DESCRIPTION WITHIN A GENERAL
PURPOSE PROGRAMMING LANGUAGE**

(52) **U.S. Cl. 717/143**

(76) **Inventor: Atsushi Kasuya, Sunnyvale, CA (US)**

(57) **ABSTRACT**

Correspondence Address:
**FENWICK & WEST LLP
SILICON VALLEY CENTER
801 CALIFORNIA STREET
MOUNTAIN VIEW, CA 94041 (US)**

(21) **Appl. No.: 11/148,126**

(22) **Filed: Jun. 7, 2005**

Publication Classification

(51) **Int. Cl.
G06F 9/45 (2006.01)**

A translation module translates a temporal description into a temporal expression. The translation module is adapted to translate a temporal description in the form of an extended syntax or a preprocessing macro to the temporal expression. The temporal expression includes a native expression of a general purpose programming language. The temporal expression may also include one or more construct functions. A parsing module parses the temporal expression to generate a data structure that represents the temporal expression without evaluating the native expression. An evaluation module evaluates the data structure to execute the temporal expression including the native expression.

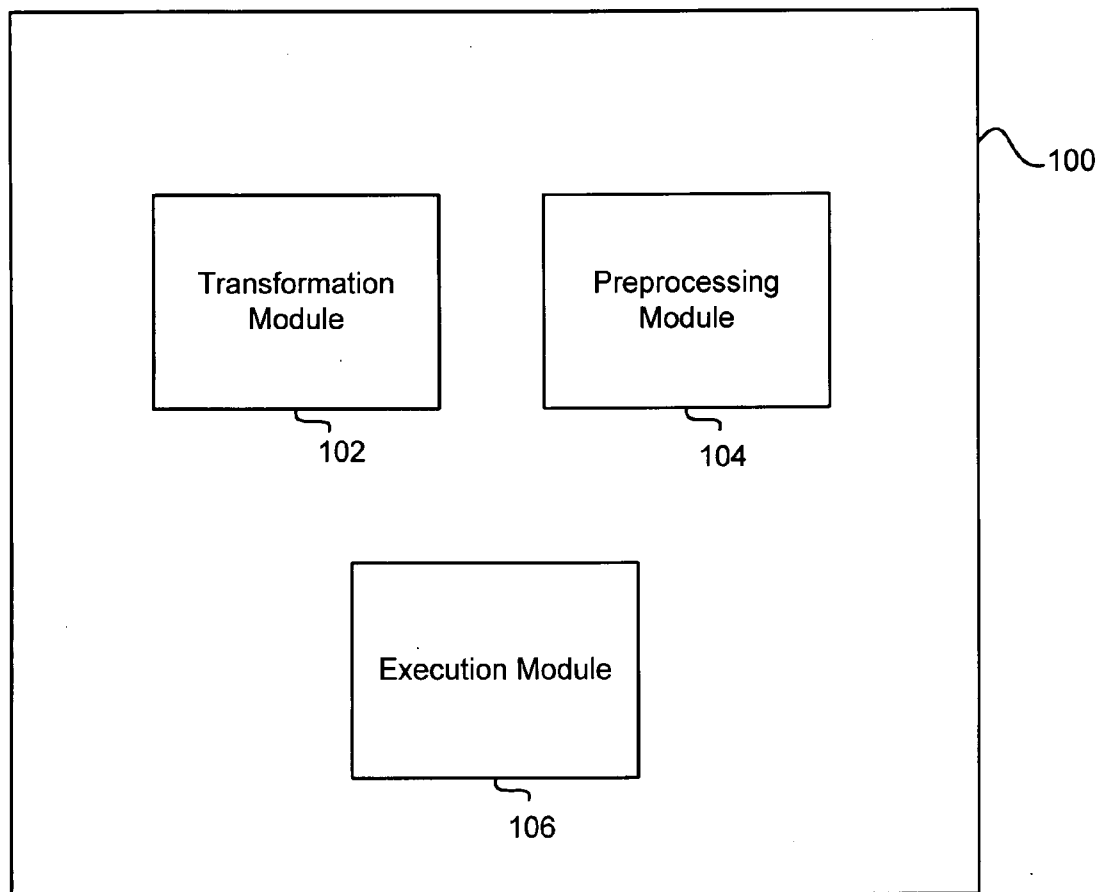
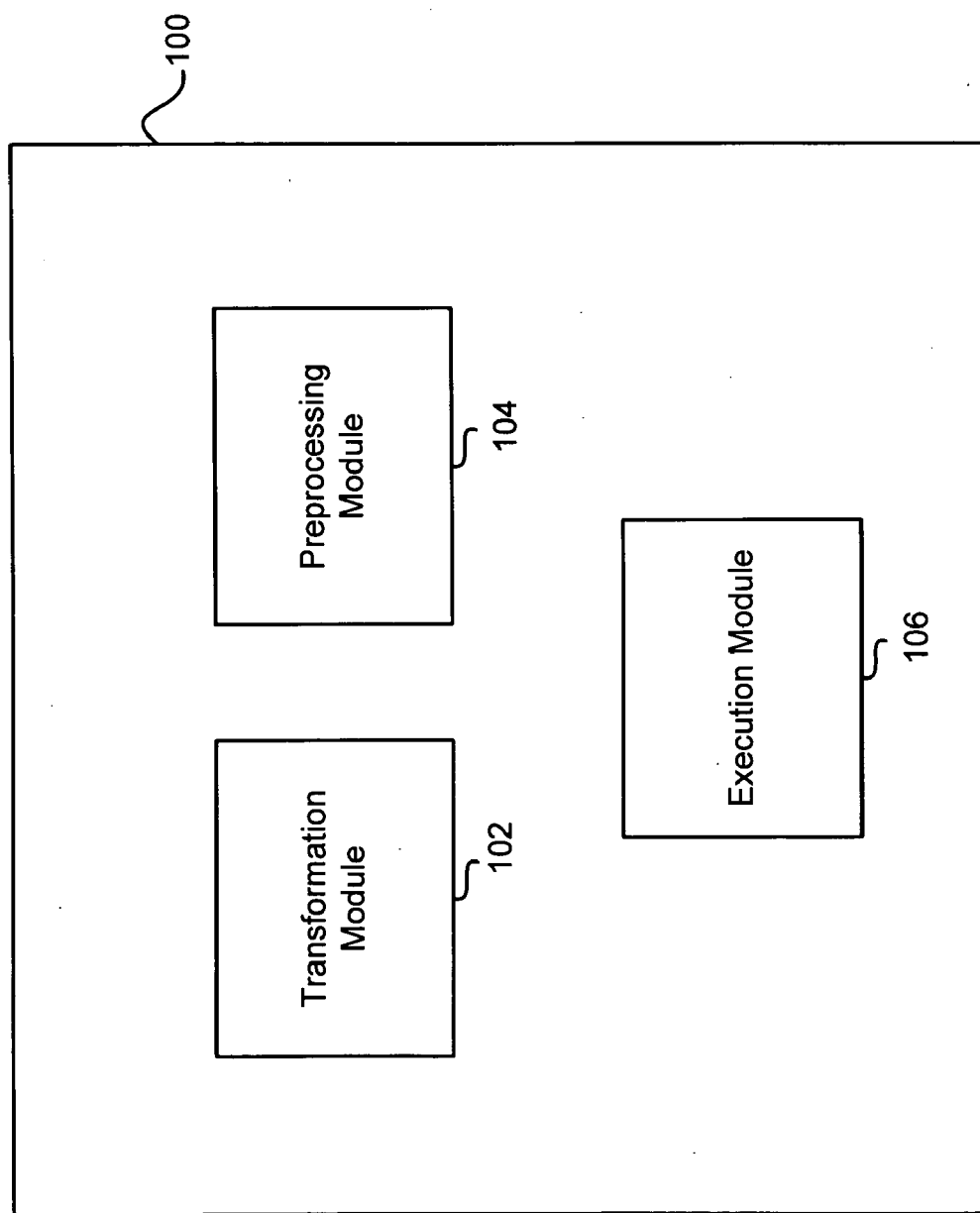


FIG. 1



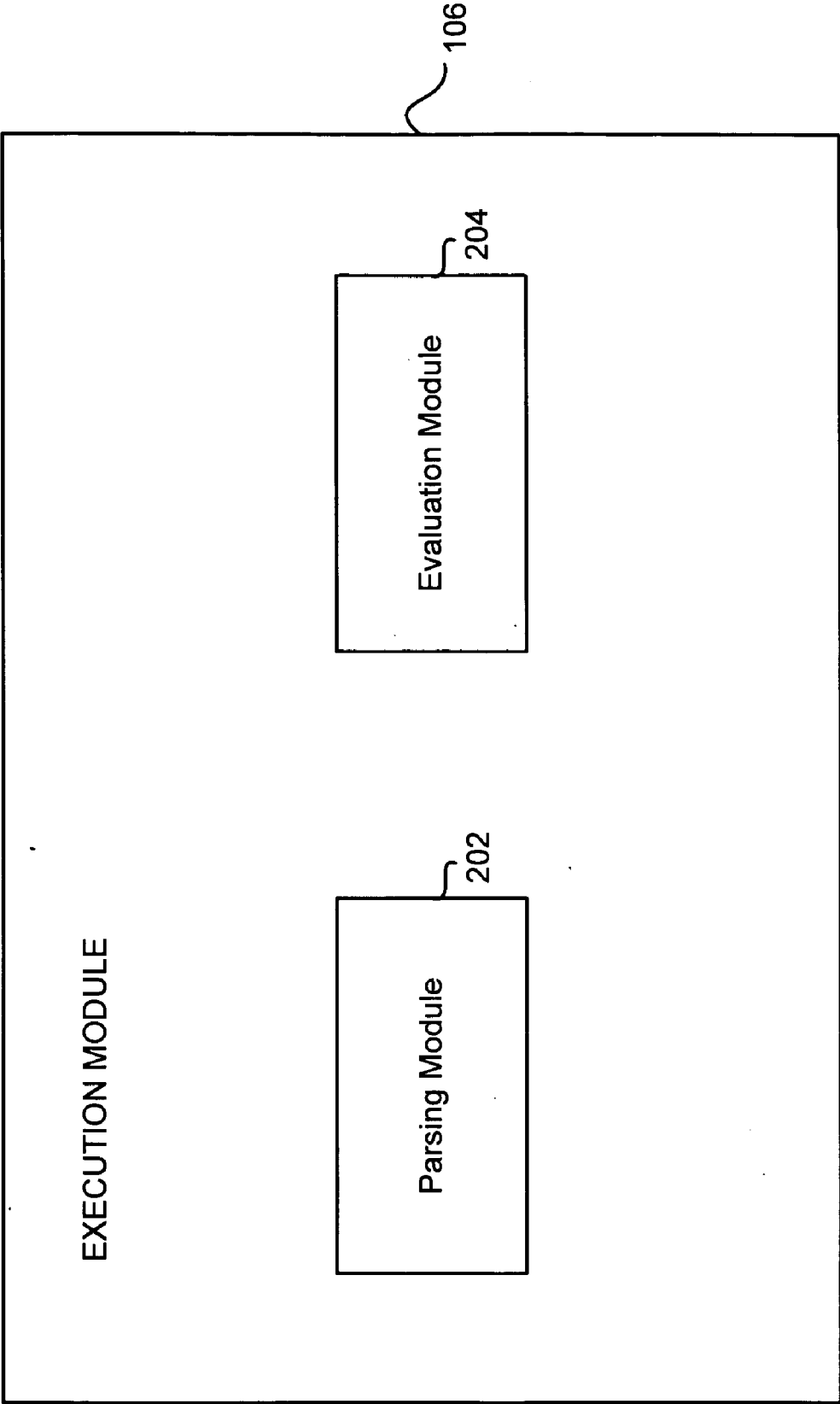


FIG. 2

FIG. 3

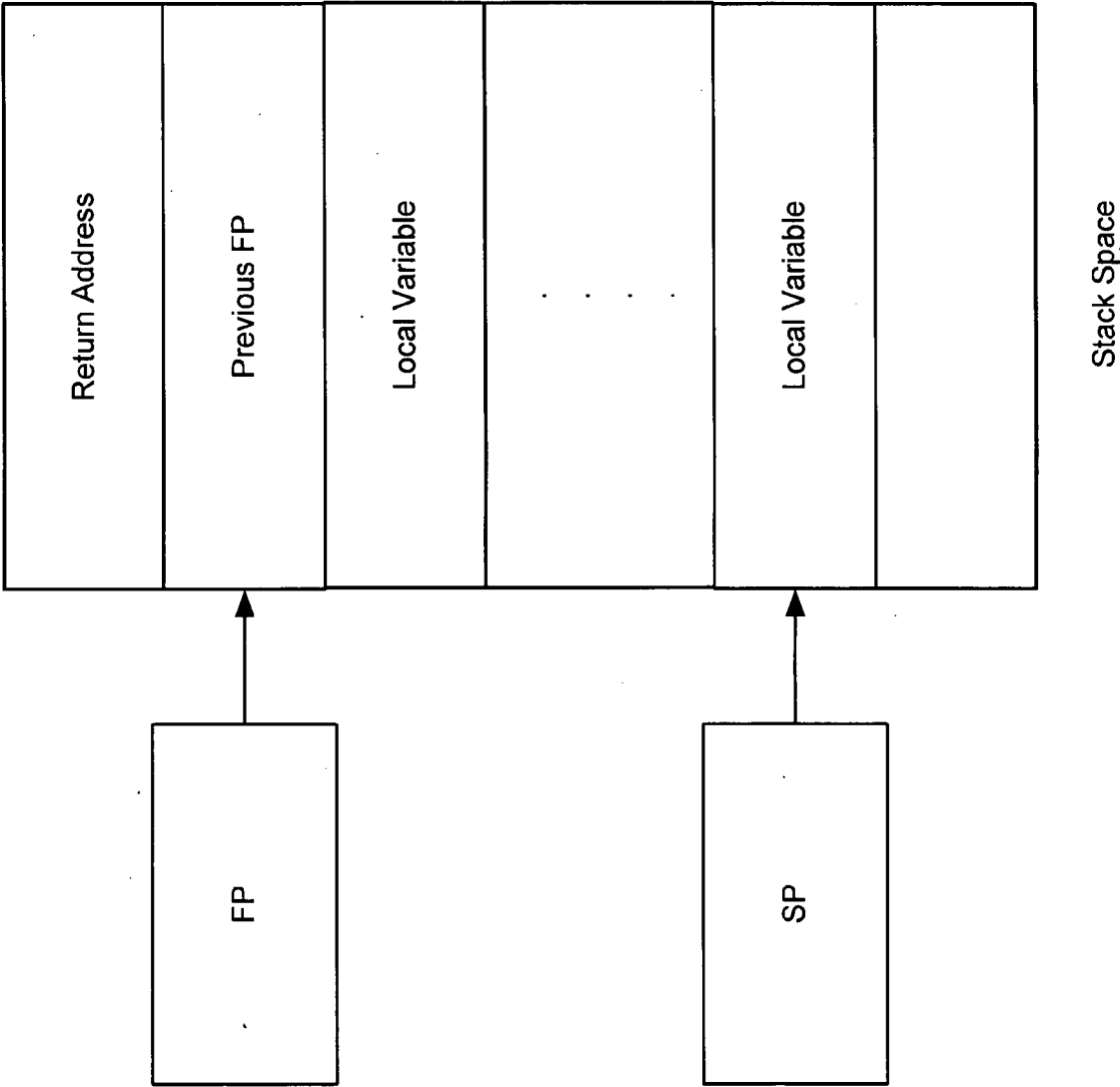


FIG. 4A

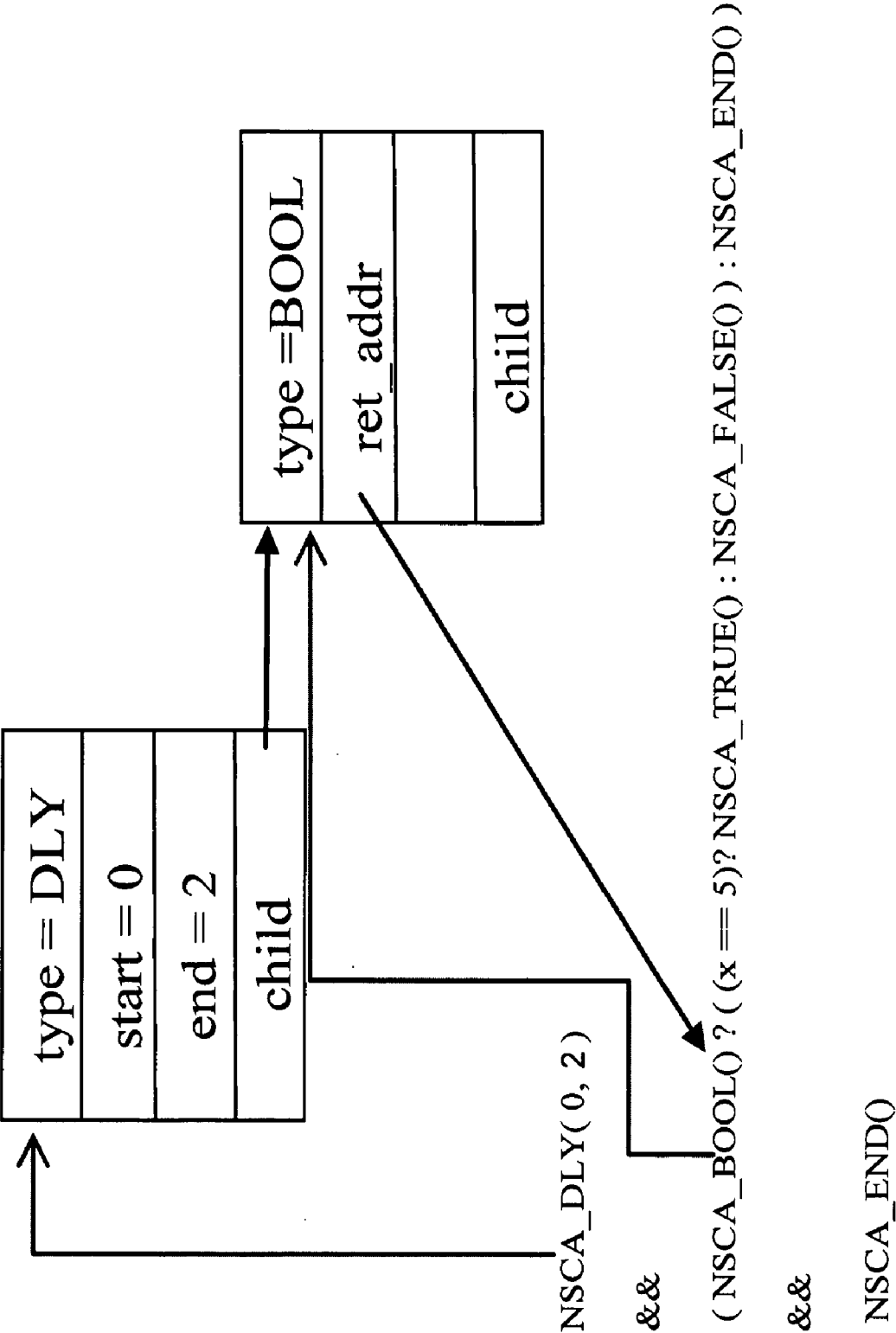
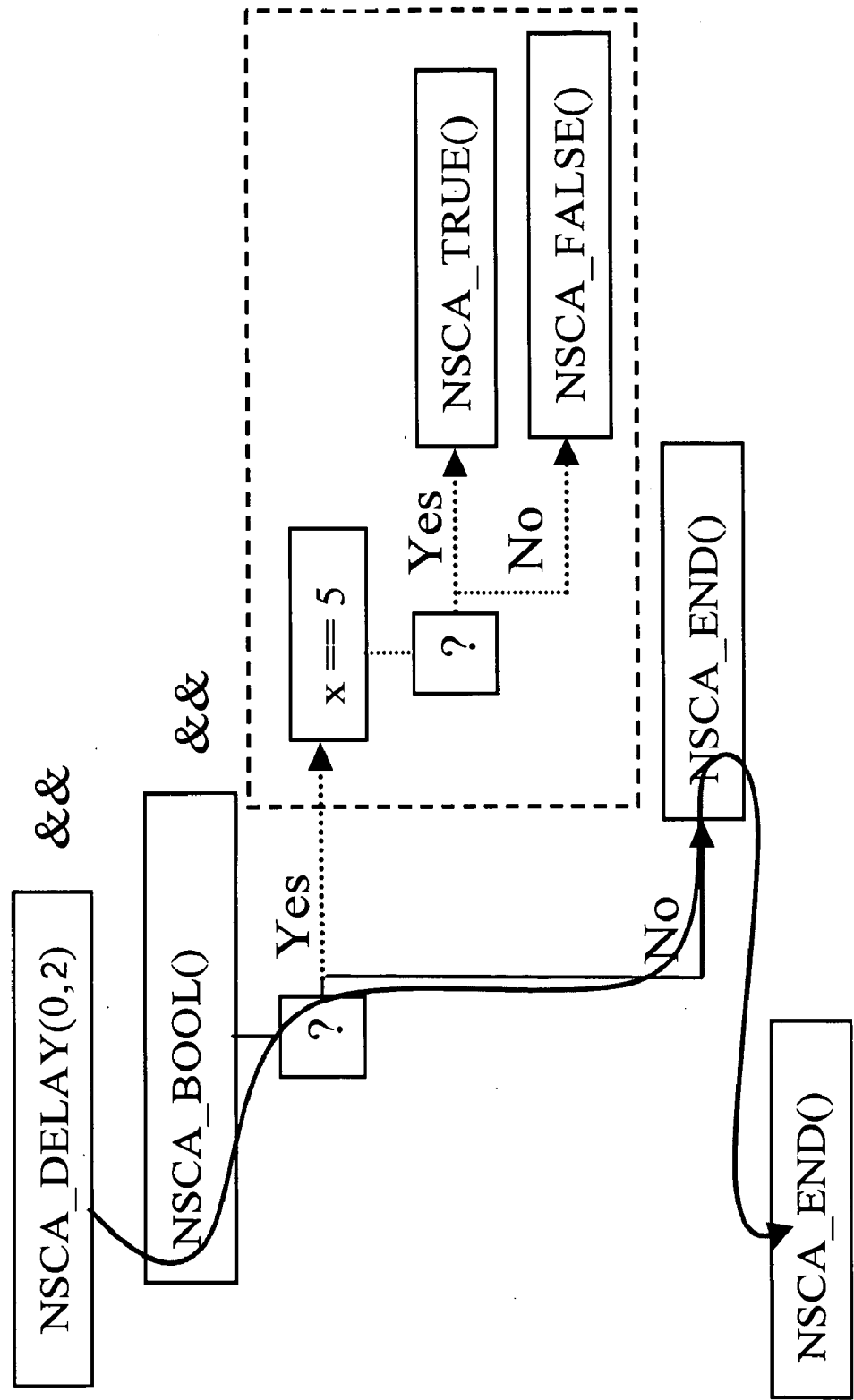


FIG. 4B



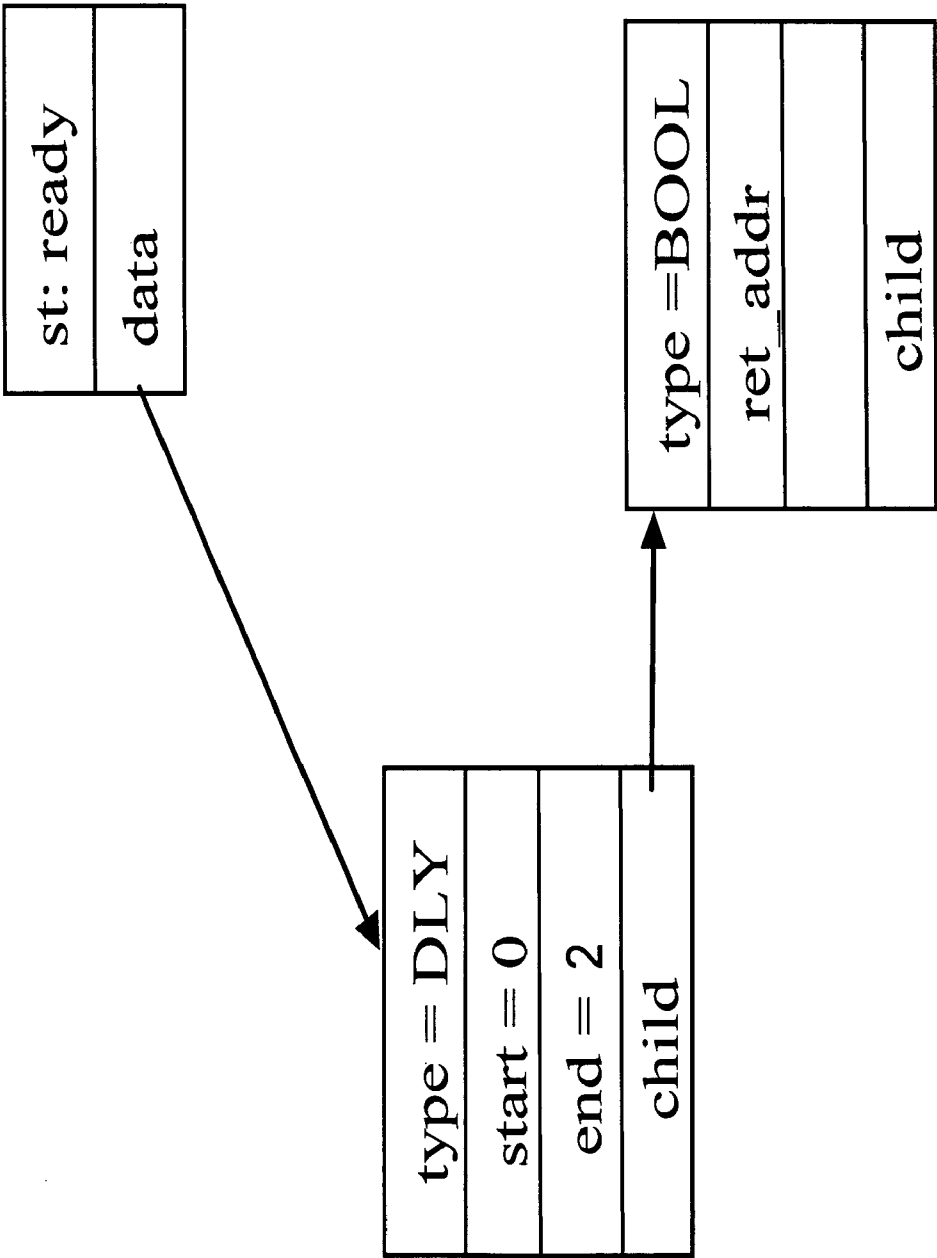


FIG. 5

FIG. 6

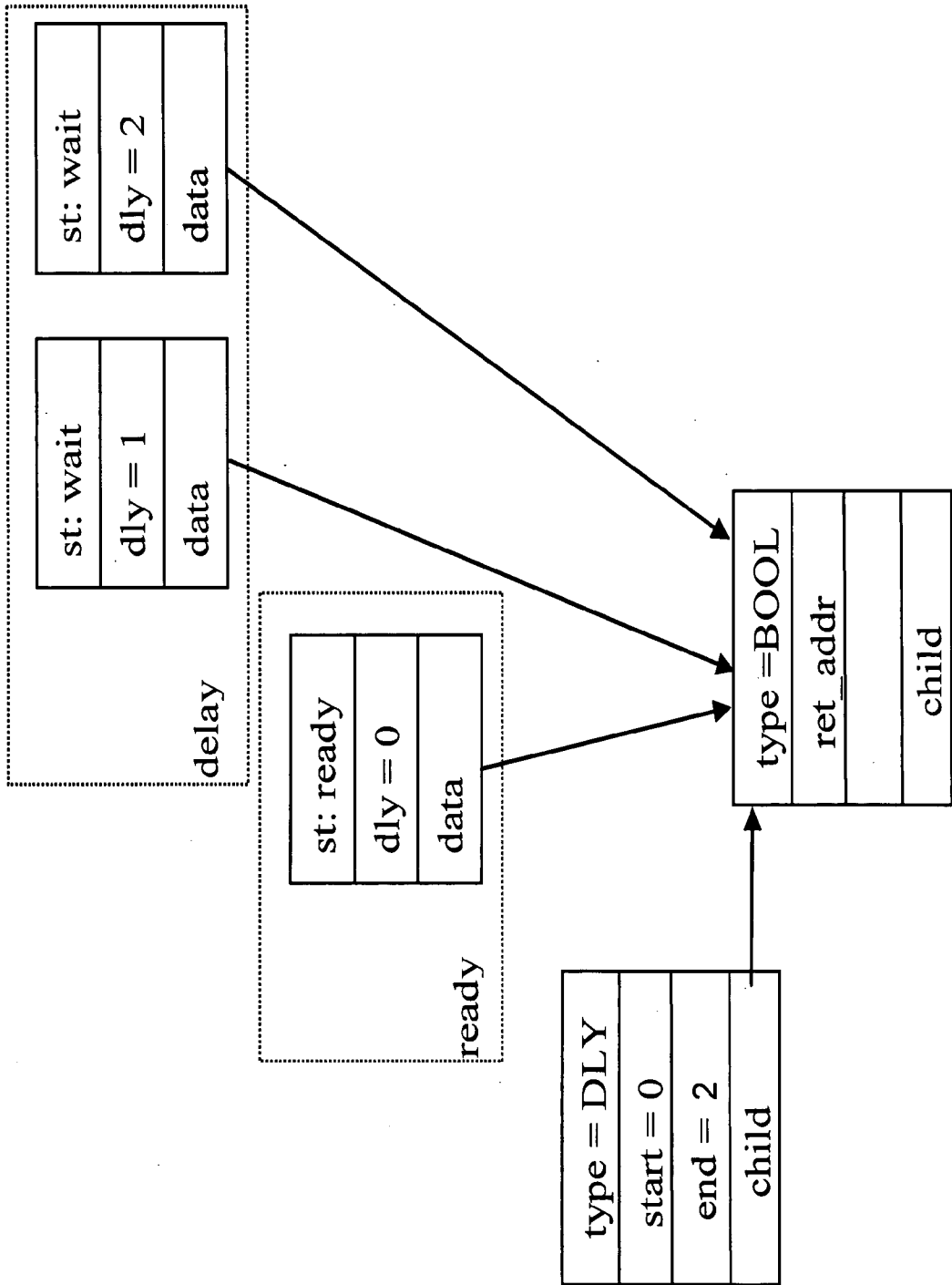


FIG. 7

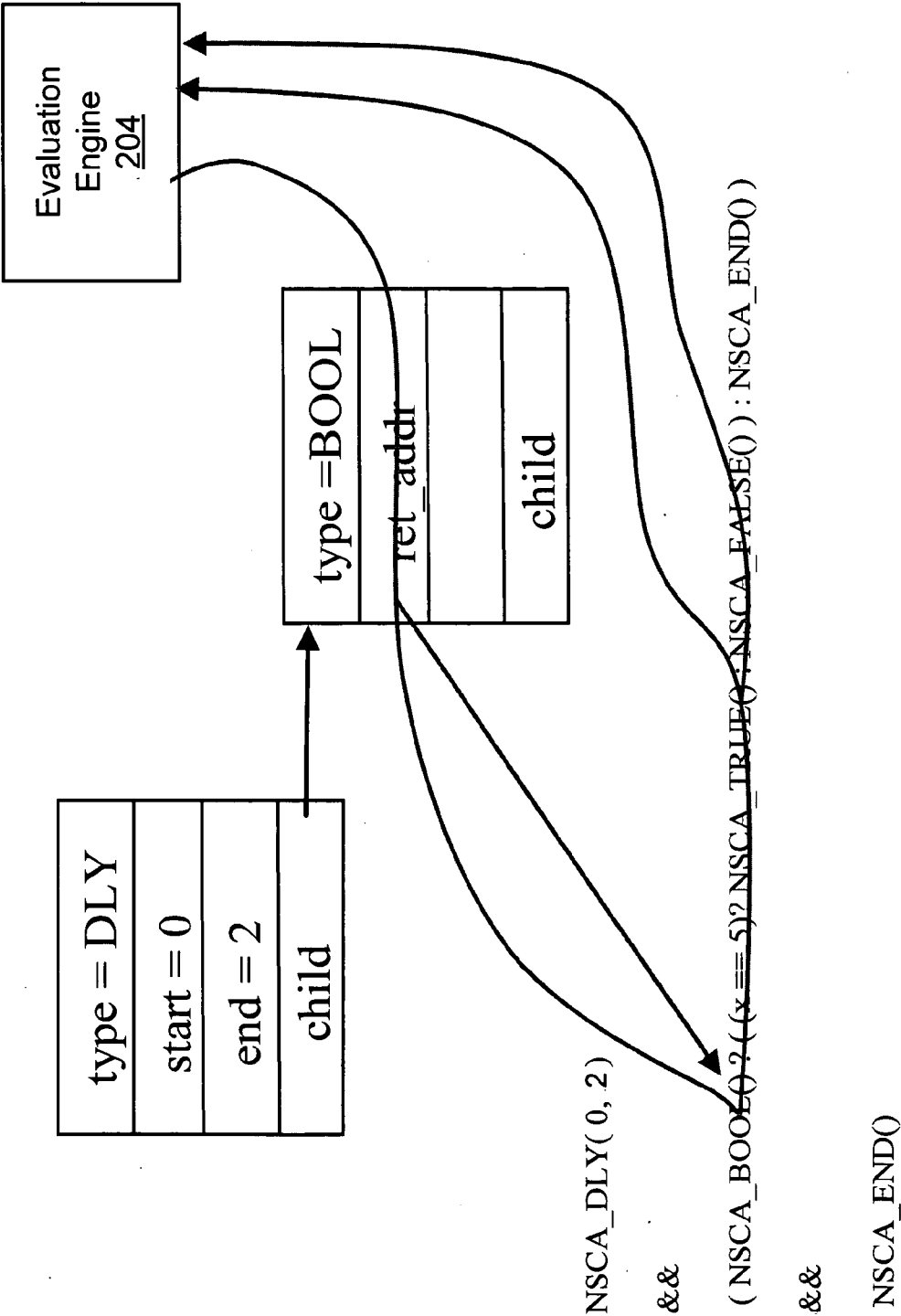


FIG. 8

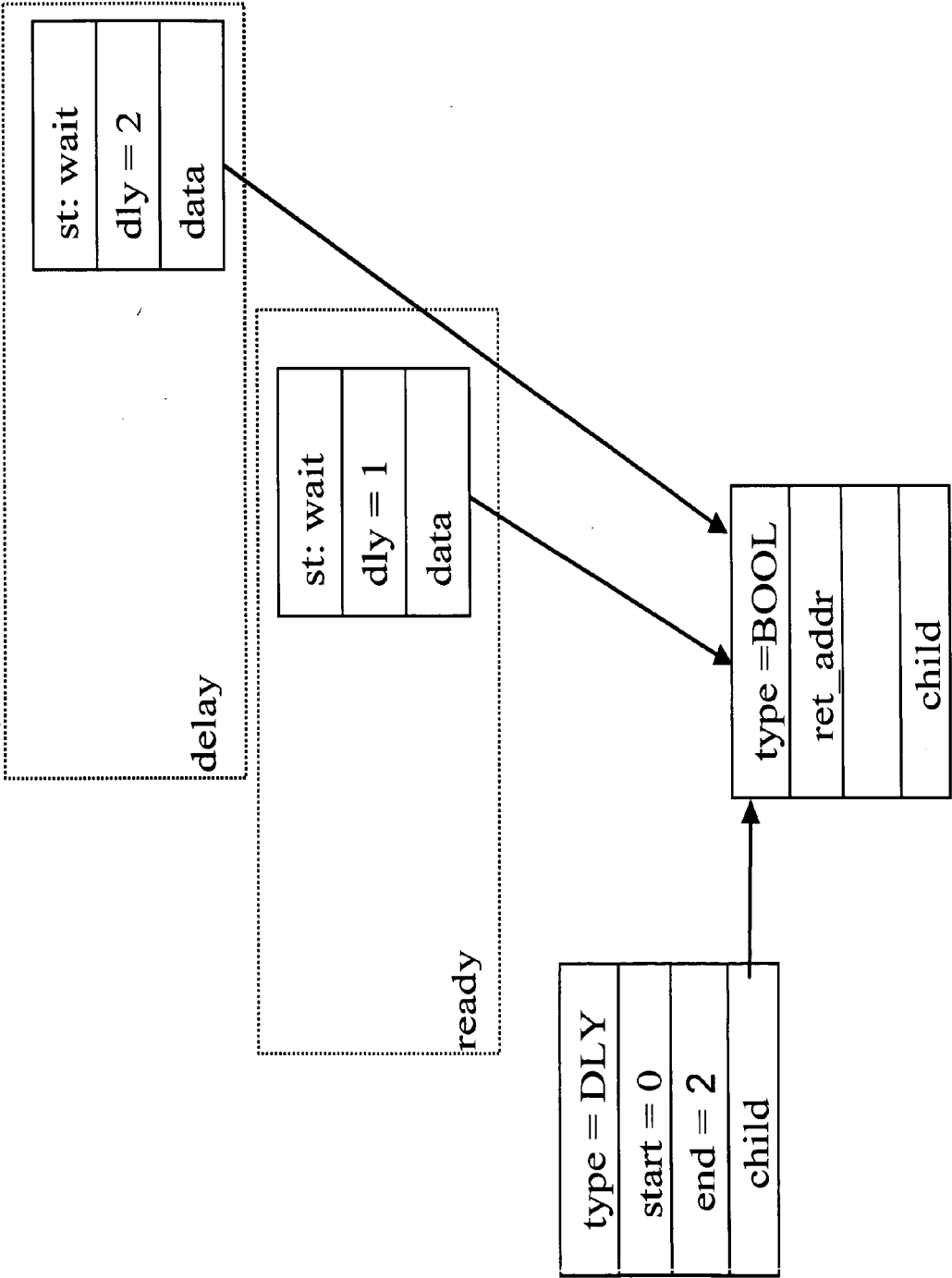


FIG. 9

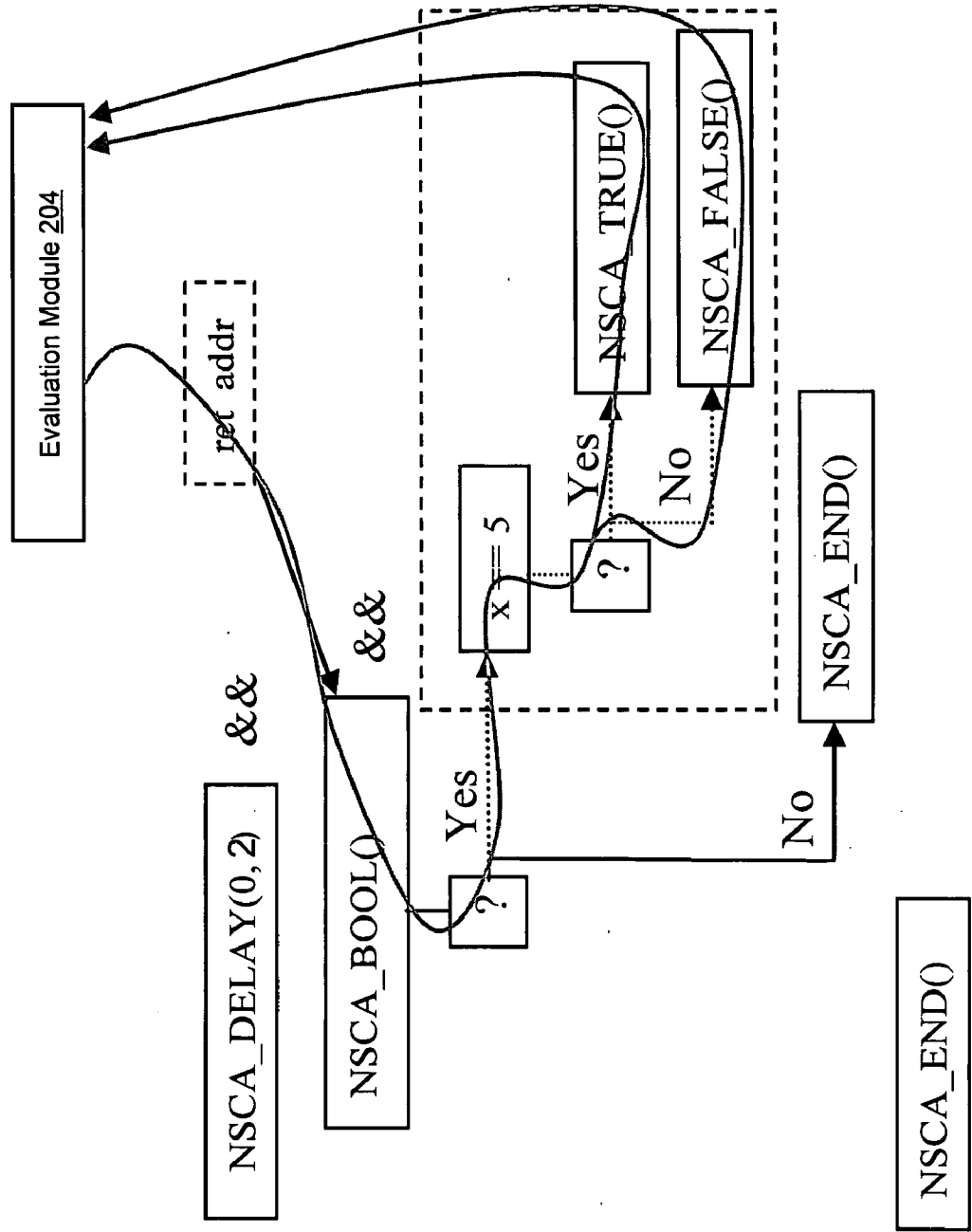


FIG. 10

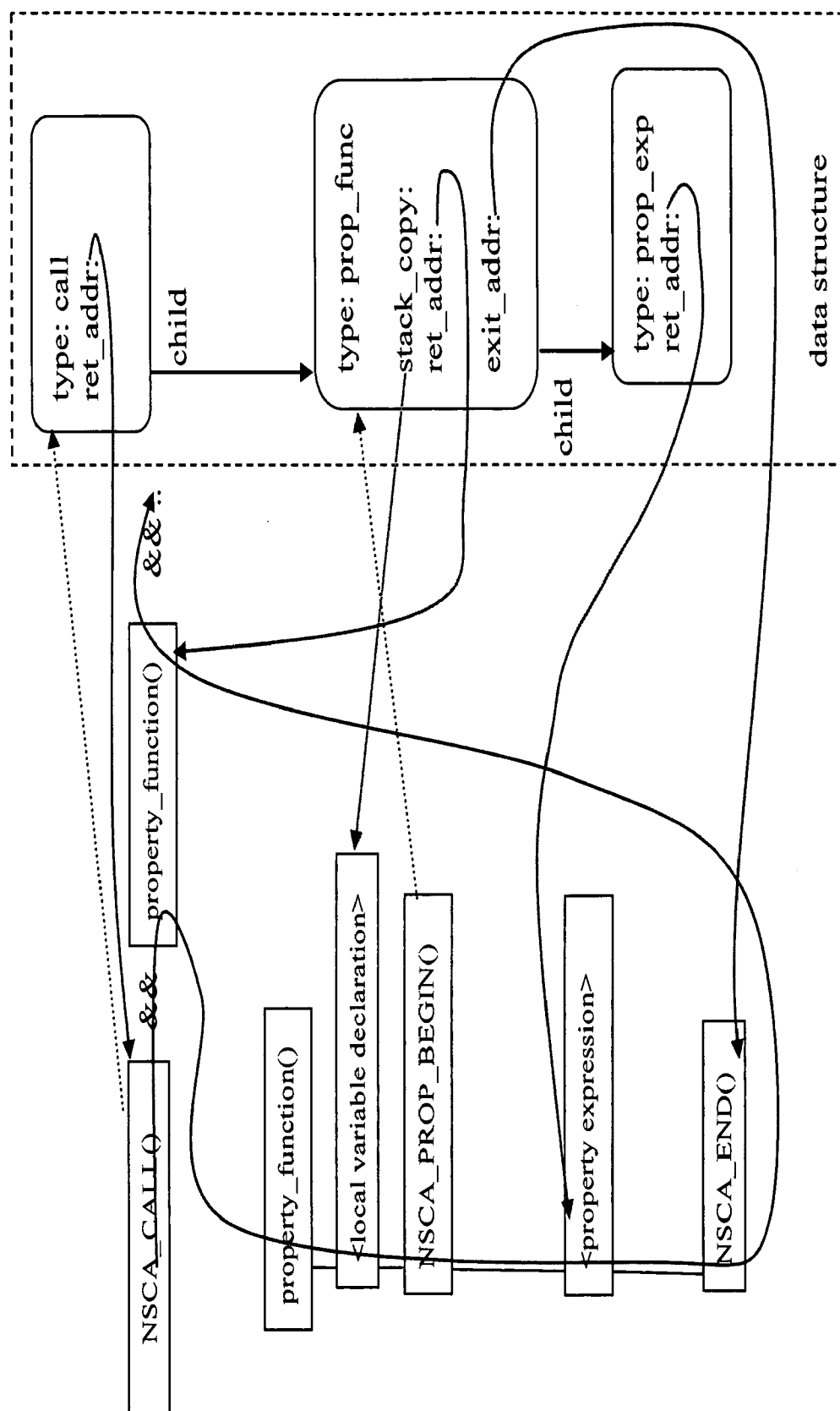
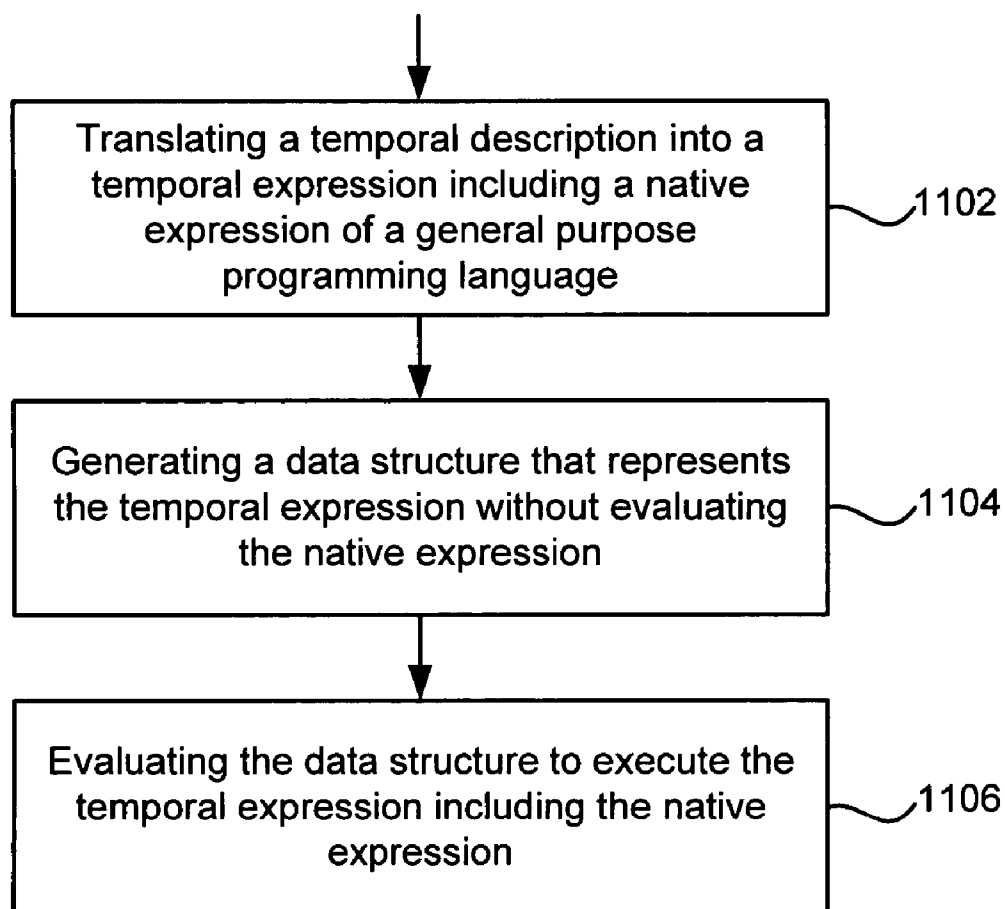


FIG. 11

EVALUATION OF A TEMPORAL DESCRIPTION WITHIN A GENERAL PURPOSE PROGRAMMING LANGUAGE

BACKGROUND OF THE INVENTION

[0001] 1. Field of the Invention

[0002] This invention pertains in general to hardware functional verification and in particular to techniques for evaluating a temporal description within a general purpose programming language.

[0003] 2. Background Art

[0004] Using a temporal description to verify a hardware design is increasingly common. Hardware description languages such as the problem statement language (PSL) are becoming standards to verify hardware designs and various electronic design automation (EDA) tools are beginning to support the evaluation of such a temporal description. For example, a hardware description language such as System Verilog uses the temporal description mechanism to check the correctness of a hardware behavior during hardware simulation. Special purpose programming languages developed for designing hardware can adopt such a new language construct (i.e., temporal description) with an additional syntax.

[0005] Another trend of hardware design is to use a general purpose programming language such as C++ to simulate a hardware behavior. In such an environment, users use a general purpose programming language with various hardware modeling support libraries to construct a simulation model to verify hardware behaviors. Because the hardware model is written in a general purpose programming language, it is easy to mix simulation codes, which are constructed using the general purpose programming language with software.

[0006] In a general purpose programming language, however, the language syntax cannot be easily changed to support hardware simulation. It is desirable for the general purpose programming language to have a mechanism to evaluate a temporal description during hardware simulation. Moreover, it is desirable that such a mechanism be able to express a temporal description within the context of the general purpose programming language such that the temporal description is constructed with the general purpose programming language's native expression, and that variables used within the temporal description are taken from the name scope of the temporal description's location.

[0007] In addition, because the temporal logic mechanism is useful in specifying the sequence of hardware behaviors, it is desirable to use it as an ordinal expression within a general purpose programming language. In such an expression, there will be a mixture of temporal constructs and ordinal expressions. To evaluate such a mixed expression, the ordinal expression is repeatedly evaluated over the program execution. However, the ordinal execution mechanism provided by general purpose programming language compilers cannot handle such a situation.

[0008] Therefore, there is a need for a technique that effectively evaluates a temporal description within a general purpose programming language.

DISCLOSURE OF INVENTION

[0009] The above need is met by a translation module that translates a temporal description into a temporal expression. The translation module is adapted to translate a temporal description in the form of an extended syntax or a preprocessing macro to the temporal expression. The temporal expression includes a native expression of a general purpose programming language. The temporal expression may also include one or more construct functions. A parsing module generates a data structure that represents the temporal expression without evaluating the native expression. Particularly, the parsing module is adapted to parse the temporal expression to generate the data structure. During the parse phase, the parsing module creates a data structure, copies a return address in a stack to a data structure for future evaluation. Then it avoids evaluating the native expression during the parse phase by controlling a construct function's return value to be false.

[0010] An evaluation module evaluates the data structure to execute the temporal expression including the native expression. For example, the evaluation module returns to the address stored in the data structure instead of the actual return address in the stack and evaluates the native expression by controlling the construct function's return value to be true. The evaluation module also evaluates a variable within the native expression from a name scope of the native expression's program location. According to an embodiment of the invention, the evaluation module creates one or more nondeterministic finite automata (NFA) state structures that include data to be evaluated at one or more clock cycles. The evaluation module stores the one or more NFA state structures in a queue. The evaluation module then retrieves a NFA state structure from the queue in response to the passing of clock cycles to evaluate the retrieved NFA state structure.

BRIEF DESCRIPTION OF THE DRAWINGS

[0011] **FIG. 1** is a high-level block diagram of a computing environment according to an embodiment of the present invention.

[0012] **FIG. 2** is a high-level block diagram illustrating modules within an execution module according to an embodiment of the present invention.

[0013] **FIG. 3** is a block diagram illustrating the configuration of an exemplary stack according to an embodiment of the present invention.

[0014] **FIGS. 4A and 4B** are block diagrams illustrating the parsing of an exemplary temporal expression according to an embodiment of the present invention.

[0015] **FIGS. 5-8** are block diagrams illustrating the evaluation of an exemplary temporal expression according to an embodiment of the present invention.

[0016] **FIG. 9** is a block diagram illustrating the execution path for an exemplary temporal expression during the evaluation phase according to an embodiment of the present invention.

[0017] **FIG. 10** is a block diagram illustrating the execution path for an exemplary expression during the evaluation phase according to an embodiment of the present invention.

[0018] **FIG. 11** is a flowchart illustrating steps performed by a computing environment according to an embodiment of the present invention.

[0019] The figures depict an embodiment of the present invention for purposes of illustration only. One skilled in the art will readily recognize from the following description that alternative embodiments of the structures and methods illustrated herein may be employed without departing from the principles of the invention described herein.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0020] **FIG. 1** is a high-level block diagram of a computing environment **100** according to an embodiment of the present invention. **FIG. 1** illustrates that the computing environment **100** includes a transformation module **102** (generally referred to as a translation module), a preprocessing module **104** (also generally referred to as a translation module), and an execution module **106**. Those of skill in the art will understand that other embodiments of the computing environment **100** can have different and/or other modules than the ones described herein. In addition, the functionalities can be distributed among the modules in a manner different than described herein.

[0021] In an embodiment of the invention, a temporal description is written within a general purpose programming language such as C++. The temporal description can be expressed with either an extended syntax or a preprocessor macro. In the case of an extended syntax, the transformation module **102** (e.g., a parser program or a compiler) parses the extended syntax and translates it into the general purpose programming language's native expression. In the case of a preprocessor macro, the preprocessing module **104** (e.g., a preprocessor such as the c preprocessor (CPP)) parses the macro and translates it into the native expression.

[0022] As a result of the translation by the transformation module **102** or by the preprocessing module **104**, the temporal description is expanded to one or more construct functions and/or native expressions, which may be connected by one or more logical "and" operators and/or "select" operators.

[0023] As an example, the temporal description that "x=5" is true within 0 to 2 cycles may be expressed with the syntax "@[0,2]x=5." This temporal description can also be represented by the macro "NSC_SEQ(0, 2, NSC_BOOL(x=5))." The transformation module **102** or the preprocessing module **104** then translates the temporal description (either in an extended syntax to be translated by the transformation module **102** or in a preprocessing module to be translated by the preprocessing module **104**) into the following temporal expression that includes one or more function calls and native expressions:

```
NSCA_DLY(0, 2) &&
(NSCA_BOOL() ? ((X == 5)? NSCA_TRUE() :
NSCA_FALSE()) : NSCA_END())
&&
NSCA_END()
```

[0024] This exemplary temporal expression includes one or more function calls and/or native expressions. Therefore, this temporal expression can be processed by a general purpose programming language's ordinal compiler. In addition, the temporal expression is a pure expression that can be

placed anywhere in a program where an expression can be placed (e.g., as a condition in an "if" statement). Since a native expression is expanded in a program location, the native expression can use a variable that can be accessed at a location of the program where the native expression is expanded. For example, the variable "x" in the native expression above can be a local variable, which is usually not accessible from outside of the program function. Thus, a variable within the native expression can be evaluated from a name scope of the native expression's program location.

[0025] In the exemplary temporal description (in the extended syntax form) "@[0,2]x=5," "@[0,2]" is a special temporal expression operator, and "x=5" is a general purpose programming language's native expression. In an embodiment, the transformation module **102** is adapted to translate "@[0,2]" into the temporal expression:

[0026] NSCA_DLY(0, 2) && . . . && NSCA_END()
Similarly, the preprocessing module **104** is adapted to translate the macro "NSC_SEQ(0, 2, . . .)" into the temporal expression:

[0027] NSCA₁₃ DLY(0, 2) && . . . && NSCA_END()

[0028] The transformation module **102** is further adapted to translate the native expression "x=5" into:

```
NSCA_BOOL() ?
((x == 5) ? NSCA_TRUE() : NSCA_FALSE()) : NSCA_END()
```

[0029] Similarly, the preprocessing module **104** is adapted to translate the macro NSC_BOOL(x=5) into:

```
NSCA_BOOL() ?
((x == 5) ? NSCA_TRUE() : NSCA_FALSE()) : NSCA_END()
```

[0030] Thus, the transformation module **102** (if the temporal description is in the extended syntax form) and the preprocessing module **104** (if the temporal description is a macro) are adapted to parse the temporal description into one or more native expressions and function calls that can be understood by a general purpose programming language's compiler. APPENDIX A provides the constructs for various syntax and their translations into a general purpose programming language such as C++. APPENDIX B provides the descriptions of the various construct fictions. Both Appendix A and Appendix B are a part of the instant specification.

[0031] In an embodiment of the invention, the execution module **106** is configured to parse the temporal expression generated by the transformation module **102** or the preprocessing module **104** to create a data structure representing the temporal expression. The execution module **106** is further configured to evaluate the created data structure to evaluate and execute the temporal expression. **FIG. 2** is a high-level block diagram of the execution module **106** according to an embodiment of the present invention. **FIG. 2** illustrates that the execution module **106** includes a parsing module **202** and an evaluation module **204**. Those of skill in the art will understand that other embodiments of the

execution module **106** can have different and/or other modules than the ones described herein. In addition, the functionalities can be distributed among the modules in a manner different than described herein.

[0032] The execution module **106** executes the generated temporal expression in two phases: a parsing phase and an evaluation phase. According to an embodiment of the invention, the parsing module **202** is adapted to parse the temporal expression to generate a data structure without evaluating the native expression within the temporal expression. Evaluating a native expression during the parsing phase is undesirable because the native expression may have side effects (e.g., the native expression “x++=5” increments x during evaluation. Thus, in the example given above, the native expression “x=5” is embedded in the temporal expression generated by the transformation module **102** or the preprocessing module **104** such that the parsing module **202** can avoid evaluating the native expression by controlling the return value of the NSCA_BOOL() function to be false. During the parsing phase conducted by the parsing module **202**, the NSCA_BOOL() function returns false (i.e., “0”) so that the “select” operator “?” elects to call the NSCA_END() function.

[0033] At the parsing phase, the parsing module **202** evaluates the temporal expression and calls one or more construct functions to generate a data structure that represents the temporal expression’s logical view. The parsing phase is processed by executing the translated temporal expression as the natural evaluation of the general purpose programming language. As the result, the execution calls various construct functions and transfers control to the parsing module **202**. According to the function types, the parsing module **202** generates a data structure representing the temporal expression’s logical view.

[0034] Various CPU architectures employ various sets of general purpose registers to evaluate the general purpose programming language, and its usage of the registers vary per CPU architecture. But in general, almost all the CPU architectures employ the stack, stack pointer register, and frame pointer register to implement the function call/return mechanism. In this embodiment, we explain the call/return activity based on a virtual CPU architecture with stack, SP (stack pointer register), and FP (frame pointer register). Implementing this scheme into a specific CPU architecture can be easily achieved because of the similarity of call/return mechanism used in various CPU architectures.

[0035] When various construction functions are called to transfer the control to the parsing module **202**, a general purpose programming language’s compiler uses one register as a frame pointer (FP) to identify the function’s local variable boundary. A stack pointer (SP) registers one or more points to the end of the stack, and the local variables are located between FP and SP. The function’s return address is placed before FP, and the previous FP value is saved in the stack where the FP register is pointing to. This configuration is shown in **FIG. 3**.

[0036] The stack-shown in **FIG. 3** grows from top to bottom, and SP points to the last valid entry in the stack space. FP points to the start point of the local variable. And the previous FP value is saved at the stack pointed to by FP. In such a configuration, the function is returned as:

[0037] SP=FP (copy FP to SP)

[0038] FP=Stack[SP—] (pop the operation from the stack and recover the previous FP value)

[0039] PC=Stack[SP—] (pop the return address to the program counter (PC))

[0040] In such an execution mechanism, the function return address can be read from the stack frame. The actual returning address can also be modified by writing a different value to the return address location in the stack frame (e.g., by using an assembly code). APPENDIX C provides four assembler functions used to execute a temporal expression. Appendix C is a part of this specification.

[0041] As discussed, the transformation module **102** or the preprocessing module **104** translates the temporal description that “x=5” is true within 0 to 2 cycles into:

```

NSCA_DLY(0, 2) &&
(NSCA_BOOL( ) ? ((X == 5)? NSCA_TRUE( ) :
NSCA_FALSE( )): NSCA_END( ))
&&
NSCA_END( )

```

[0042] During the parsing phase, the parsing module **202** calls the construct functions in the following order:

[0043] NSCA_DLY(0, 2)

[0044] NSCA_BOOL()

[0045] NSCA_END()

[0046] NSCA_END()

[0047] However, the parsing module **202** does not evaluate the expression “((X=5)? NSCA_TRUE() : NSCA_FALSE())” because the NSCA_BOOL() function returns false or “0” during the parsing phase. After the parsing module **202** completes the parsing phase, it evaluates the temporal expression and constructs a data structure. An exemplary data structure constructed by the parsing module **202** for the exemplary temporal expression is shown in **FIG. 4A**.

[0048] As shown in **FIG. 4A**, the NSCA_DLY() function creates a structure **402** having a DLY (delay) type, and the structure holds the start and end cycle value. The structure is coupled to its child until the corresponding NSCA_END() function is called. The NSCA_BOOL() function creates a structure **404** having a BOOL type. The return address from the NSCA_BOOL() function is kept in the structure for future evaluation.

[0049] If the parsing module **202** detects that the NSCA_END() function closes the top level structure, it knows that the entire temporal expression has been parsed. Accordingly, the evaluation module **204** can evaluate the constructed data structure that represents the temporal expression.

[0050] **FIG. 4B** illustrates the execution path during the parsing phase. As shown, the NSCA_BOOL() function returns false or “0” in response to capturing the return address such that the native expression “x=5” is not evaluated 450.

[0051] In an embodiment of the invention, the evaluation module 204 evaluates the temporal expression using an NFA mechanism. Specifically, the evaluation module 204 constructs another data structure representing an active state within an NFA. In such an implementation, the evaluation module 204 creates a bubble (i.e., the NFA state structure) that points to the top data structure. The evaluation module 204 places the bubble in an active queue to be evaluated. The evaluation module 204 then processes one or more bubbles in the active queue. The evaluation module 204 can place a bubble in a delay queue if the bubble is to be delayed for one or more cycles. APPENDIX D provides a description of the NFA mechanism for evaluating a temporal expression. Appendix D is a part of this specification.

[0052] FIGS. 5-8 illustrate the evaluation mechanism for the exemplary temporal expression. As shown in FIG. 5, the evaluation module 204 creates an initial bubble 502 pointing to the DLY type structure 510. As shown in FIG. 6, in response to the evaluation module 204 evaluating the initial bubble 502 pointing to the DLY type structure, with start=0 and end=2, the initial bubble produces three bubbles 602, 604, 604, with 0 to 2 delays. The bubble with 0 delay stays in a ready queue for immediate evaluation by the evaluation module 204. Other bubbles are placed in a delay queue and reactivated (e.g., placed in the ready queue) in response to the passing of clock cycles.

[0053] In FIG. 7, the evaluation module 204 evaluates the ready bubble 510 pointing to the BOOL type structure. Since the evaluation module 204 is run within the last NSCA_ENDO() function 710, the evaluation module 204 can execute the Boolean expression by returning to the address stored in the "ret_addr" field of the data structure, with the return value of "1." This has the same effect as returning from the NSCA_BOOL() function within the expression, with the return value of "1." The evaluation module 204 then proceeds to the "select" operator "?" and evaluates the native expression "x=5." Depending on the result of this evaluation, the evaluation module 204 calls the NSCA_TRUE() function or the NSCA_FALSE() function and obtains the result condition. The evaluation module 204 records the result condition to be used as the final evaluation result.

[0054] As shown in FIG. 8, at the next cycle, the subsequent bubbles in the wait queue become active, and the evaluation module 204 executes the same Boolean evaluation. The evaluation module 204 completes the evaluation if bubbles are completed and removed from the ready queue and the delay queue. The evaluation module 204 thus returns to the last NSCA_END() function address with the proper evaluation result.

[0055] FIG. 9 illustrates the execution path for the exemplary temporal expression during the evaluation phase. As shown, during the evaluation phase, the NSCA_BOOL() function's return address is kept as the evaluation address of the native expression "x=5" 450. If the evaluation module 204 evaluates the Boolean expression, it sets the return address and returns to the evaluation point. The result is transferred back to the evaluation module 204 after the evaluation.

[0056] APPENDIX E provides the various functions that may be executed during the parsing phase and the evaluation phase. Appendix E is a part of this specification.

[0057] With this evaluation mechanism, a temporal expression can be embedded as an ordinal expression in a general purpose programming language. A native expression used within the temporal expression can use a variable within the scope of the embedded point. The text below provides an exemplary code in a general purpose programming language.

```
void foo( ) {
    int x, y ;
    ..
    if( @[3,5] data == x ) {
        ..
    }
}
```

[0058] In the exemplary code above, the temporal description "@[3,5] data=x" can be used as a condition of the "if" statement, and the local variable "x" can be used in a temporal expression. The transformation module 102 or the preprocessing module 104 is configured to translate this exemplary code into the form of "NSCA_DLY(3, 5) && . . . && NSCA_END()" that is acceptable in a general purpose programming language's syntax.

[0059] In another example, the transformation module 102 or the preprocessing module 104 is adapted to translate a property expression and a sequence expression into the following form:

[0060] NSCA_CALL() && (<function call>) && NSCA_CALL_END()

[0061] The corresponding property function has the structure of:

```
void property_name ( <arguments> ) {
    <local_variable_declarations>
    NSCA_PROP_BEGIN( ) ;
    <property expression>
    NSCA_PROP_END ( ) ;
}
```

[0062] The corresponding sequence function has the structure of:

```
int <sequence_name> ( <arguments> ) {
    <variable declaration>
    NSCA_SEQ_BEGIN( ) ;
    <sequence expression>
    NSCA_SEQ_END( ) ;
}
```

[0063] During the parsing phase, if the NSCA_CALL() function is called, the data structure that represents the subroutine call is created, and the return address of the NSCA_CALL() is recorded as the evaluation point of the subroutine. The NSCA_CALL() function then returns "1," and the evaluation module 204 calls the corresponding sub-function.

[0064] Within the sub-function, in response to the local variable being created in the stack frame, either the NSCA-

_PROP_BEGIN() function or the NSCA_SEQ_BEGIN() function is called. Within the function, a new structured (coupled under the call structure created by the NSCA_CALL() function) and a copy of the local variable are created. This is accomplished by creating a copy of the current stack call frame (e.g., a copy from the FP to the current SP). The parsing module 202 then parses the property expression or the sequence expression within the corresponding function during the parsing phase. The parsing module 202 further constructs the property or sequence structure under the current function structure.

[0065] FIG. 10 illustrates the execution path for the property or sequence expression during the evaluation phase. During the parsing phase, the parsing module 202 creates a data structure that represents the property or sequence expression. The data structure is linked to represent the hierarchy of the expression. Each data structure has the type field to show the type of the data structure and holds the return address to exit from the hierarchy. For the property function structure, the stack frame copy is held to keep the local variable's value for iterative execution.

[0066] FIG. 11 is a flowchart illustrating steps performed by the computing environment 100 according to an embodiment of the invention. The figure omits some possible steps. In addition, some embodiments perform different steps in addition to, or instead of, the ones described herein. Furthermore, the order of the steps can vary from that described herein.

[0067] A translation module translates 1102 a temporal description into a temporal expression. For example, the transformation module 102 translates a temporal description in the form of an extended syntax into the temporal expression. Alternatively, the preprocessing module 104 translates a temporal description in the form of a preprocessing macro into the temporal expression. The temporal expression includes a native expression of a general purpose programming language. The temporal expression may also include one or more construct functions.

[0068] The parsing module 202 generates 1104 a data structure that represents the temporal expression without evaluating the native expression. Specifically, the parsing module 202 is adapted to parse the temporal expression to generate the data structure. To generate the data structure, the parsing module 202 stores a return address in a stack and avoids evaluating the native expression by controlling a construct function's return value to be false.

[0069] The evaluation module 204 evaluates 1106 the data structure to execute the temporal expression including the native expression. For example, the evaluation module 204 returns to the return address stored in the stack and evaluates the native expression by controlling the construct function's return value to be true. The evaluation module 204 also evaluates a variable within the native expression from a name scope of the native expression's program location. According to an embodiment of the invention, the evaluation module 204 creates one or more NFA state structures that include data to be evaluated at one or more clock cycles. The evaluation module 204 stores the one or more NFA state structures in a queue. The evaluation module 204 then retrieves a NFA state structure from the queue in response to the passing of clock cycles to evaluate the retrieved NFA state structure.

[0070] The above description is included to illustrate the operation of the preferred embodiments and is not meant to limit the scope of the invention. Where the description refers to "one" embodiment and/or "another" embodiment, it will be understood that the description is not necessarily referring to different embodiments. The scope of the invention is to be limited only by the following claims. From the above discussion, many variations will be apparent to one skilled in the relevant art that would yet be encompassed by the spirit and scope of the invention.

Appendix A

[0071] APPENDIX A provides the constructs for various syntax and their translations into a general purpose programming language such as C++.

Property Description

[0072] A property (the temporal condition over a design) is described as a property description. The syntax of a property is:

```
BNF:
'nsc_property' property_name '(' [ formal_args ] ')' '{'
    local_variable_decl
    property_expr
    '}'
property_name :=
    Identifier
```

[0073] The property is translated into a function with two fictions NSCA_PROP_BEGIN() and NSCA_PROP_END() that specify the beginning and end of the property. For example, a property is expressed as a function as:

```
void req_gnt_prop ( ) {
    int x ;
    NSCA_PROP_BEGIN( ) ;
    /* property expression, not shown now */
    NSCA_PROP_END ( ) ;
}
```

Assert Statement

[0074] The following syntax provides an assert statement to activate the property.

```
BNF:
assert_statement :=
'nsc_assert' property_name '(' [ actual_args ] ')'
```

For example:

[0075] nsc_assert req_gnt_prop() ;

[0076] The property is activated for evaluation if the function is invoked as a thread in a computing environment. The assert statement above will be translated into:

```

this_class c = this ;
sc_spawn( sc_bind( &req_gnt_prop, c ) ) ;

```

Property Expression

[0077] A property expression expresses the property of the target design. The syntax of a property expression is:

BNF:

```

property_expr :=
sequence_expr
|
'(' property_expr ')'
|
'nsc_not' property_expr
|
imply_property_expr 'nsc_or' imply_property_expr
|
imply_property_expr 'nsc_and' imply_property_expr
|
'if' '(' expression ')' property_expr [ 'else' property_expr ']'
|
property_instance
imply_proprty_expr :=
sequence_expr '|->' property_expr
|
sequence_expr '|=>' property_expr
property_instance :=
property_name '(' [ act_arg_list ] ')'

```

[0078] The negation of property “nsc_not” can be expressed with the macro:

[0079] NSC_NOT(property_expr)

This is translated as:

[0080] NSCA_NOTO && (property_expr) &&
NSCA_END()

[0081] The syntax of the “and” operation is:

BNF:

```

property_expr :=
..
imply_property_expr 'nsc_and' imply_property_expr
..

```

This expression can be specified with a macro:

[0082] NSC_PROP_AND(X, Y)

This expression is translated as:

[0083] NSCA_PAND() && (X) NSCA_NEXT() && (Y)
&& NSCA_END()

[0084] The syntax of “or” operation is:

BNF:

```

sequence_expr :=
..
imply_property_expr 'nsc_or' imply_property_expr
..

```

This expression can be specified with a macro:

[0085] NSC_PROP_OR (X, Y)

This expression is translated as:

[0086] NSCA_POR() && (X) NSCA_NEXT() && (Y)
&& NSCA_END()

[0087] The syntax of an implication property is:

BNF:

```

imply_property_expr :=
sequence_expr '|->' property_expr
|
sequence_expr '|=>' property_expr

```

The first line with ‘|->’ is an overlapped implication that the match of the first property_expr is the starting point of the second property_expr evaluation. The second line with ‘|=>’ is a non-overlapped implication that the evaluation of the second property_expr starts one cycle after the match of the first property_expr.

[0088] This can be written with a macro:

```

NSC_IMPLY( sequence_expr, sequence_expr )
NSC_NOIMPLY(sequence_expr, sequence_expr )

```

[0089] Using the “not” operation, this macro can be used as:

```

NSC_IMPLY( sequence_expr, NSC_NOT(sequence_expr) )
NSC_NOIMPLY(sequence_expr, NSC_NOT(sequence_expr) )

```

[0090] The final translation for each implication is respectively:

```

NSCA_IMP( ) && property_expr && NSCA_NEXT( ) &&
property_expr && NSCA_END( )
NSCA_NOIMP( ) && property_expr && NSCA_NEXT( ) &&
property_expr && NSCA_END( )

```

Sequence Expression

[0091] The syntax of a sequence is:

BNF:
<pre>nsc_sequence_expression := 'nsc_sequence' [clock_event_spec] '(' nsc_sequence_item { nsc_sequence_item } ') clock_event_spec := '@' '(' [edge_spec] clock_event_expression ')'</pre>

This can be specified with a macro:

[0092] NSC_SEQUENCE(clock_event_expr, sequence_item_list)

In the macro, if there is more than one sequence_item, they connected with the “&&” operator.

If a clocking event is not needed, 0 (zero) is used.

[0093] This is translated as:

<pre>NSCA_SEQUENCE(clocking_event_expr) && (sequence_item_list) && NSCA_END()</pre>
--

Sequence Item

[0094] A sequence item is a unit to construct a sequence. Multiple items can be listed to specify the flow of a temporal sequence. The syntax of a sequence item is:

BNF:
<pre>nsc_sequence_item := [cycle_delay_range] sequence_expr cycle_delay_range := '@' constant_expression '@' '[' constant_range '[' constant_range := constant_expression ':' constant_expression constant_expression ':' '\$'</pre>

An expression “@[m,n] X”, where m and n are constant integer and X is a sequence_expr, means the sequence matches if X becomes true between m and n cycles.

The same sequence item can be expressed with a macro:

[0095] NSC_SEQ(m, n, sequence_expr)

This expression is translated to:

[0096] NSCA_DLY(m, n) && (X) && NSCA_END()

If there is more than one sequence item, they are connected with the “&&” operator.

Sequence Expr

[0097] Sequence_expr is a primitive sequence element. The syntax of sequence_expr is:

BNF:
<pre>sequence_expr := systemc_simple_expression [boolean_abbrev] '(' systemc_simple_expression { ',' sequence_match_item } ')' [boolean_abbrev] '(' sequence_expr { ',' sequence_match_item } ')' [sequence_abbrev] sequence_expr 'nsc_and' sequence_expr sequence_expr 'nsc_or' sequence_expr sequence_expr 'nsc_intersect' sequence_expr sequence_expr 'nsc_within' sequence_expr systemc_simple_expression 'nsc_throughout' sequence_expr 'nsc_first_match' '(' sequence_expr { , sequence_match_item } ')' sequence_instance sequence_match_item := operator_assignment inc_or_dec_expression function_call boolean_abbrev := consecutive_repetition non_consecutive_repetition goto_repetition sequence_abbrev := consecutive_repetition consecutive_repetition := '[' '*' const_or_range_exp '[' non_consecutive_repetition := '[' '=' const_or_range_exp '[' goto_repetition := '[' '->' const_or_range_exp '[' const_or_range_exp := constant_expression constant_range sequence_instance := sequence_identifier ['(' actual_args ')']</pre>

Boolean Expression

[0098] A Boolean expression is expressed with the NSC_BOOL macro:

[0099] NSC_BOOL(X)

[0100] This macro is expanded as the function form:

<pre>(NSC_BOOL() ? ((X) ? NSCA_TRUE() : NSCA_FALSE()) : NSCA_END())</pre>

Sequence Match Item

[0101] A sequence match item is expressed with the NSC_MATCH macro:

[0102] NSC_MATCH(X)

[0103] This macro is expanded as the function form:

```
( NSCA_MATCHED( ) ?
  ( ( X ) ? NSCA_MDONE( ) : NSCA_MDONE( ) ) : NSCA_END( )
)
```

Consecutive Repetition

[0104] For an expression “X[*m:n]”, X is a sequence and m and n are integer constants. For example, n can be -1 that represents an infinite cycle. The consecutive repetition specifies finitely many iterative matches of X, with a delay of one cycle from the end of one match to the beginning of the next match.

This expression can be specified with a macro:

[0105] NSC_CREP(m, n, X)

This expression is translated as:

[0106] NSCA_CREP(m, n) && (X) && NSCA_END()

Goto Repetition

[0107] For an expression “X[->m:n]”, X is a sequence and m and n are integer constants. For example, n can be -1 that represents an infinite cycle. The goto repetition specifies finitely many iterative matches of X with a delay of one or more cycles from the end of one match to the beginning of the next match. The overall repetition sequence matches at the last iterative matches of the operand.

This expression can be specified with a macro:

[0108] NSC_GOTO(m, n, X)

This expression is translated as:

[0109] NSCA_GOTO(m, n) && (X) && NSCA_END()

Non-Consecutive Repetition

[0110] For an expression “X[=m:n]”, X is a sequence and m and n are integer constants. For example, n can be -1 that represents an infinite cycle. The non-consecutive repetition specifies finitely many iterative matches of X with a delay of one or more cycles from the end of one match to the beginning of the next match and no match of the sequence strictly in between.

This expression can be specified with a macro:

[0111] NSC_NREP(m, n, X)

This expression is translated as:

[0112] NSCA_NREP(m, n) && (X) && NSCA_END()

AND Operator

[0113] The syntax for the “and” operation of two sequences is:

```
BNF:
sequence_expr :=
..
sequence_expr 'nsc_and' sequence_expr
..
```

This expression can be specified with a macro:

[0114] NSC_AND(X, Y)

This expression is translated as:

[0115] NSCA_AND() && (X) NSCA_NEXT() && (Y) && NSCA_END()

OR operator

[0116] The syntax for the “or” operation of two sequences is:

```
BNF:
sequence_expr :=
..
sequence_expr 'nsc_or' sequence_expr
..
```

This expression can be specified with a macro:

[0117] NSC_OR (X, Y)

This expression is translated as:

[0118] NSCA_OR() && (X) NSCA_NEXT() && (Y) && NSCA_END()

INTERSECT Operator

[0119] The syntax for the “or” operation is:

```
BNF:
sequence_expr :=
..
sequence_expr 'nsc_intersect' sequence_expr
..
```

This expression can be specified with a macro:

[0120] NSC_INTERSECT (X, Y)

This expression is translated as:

[0121] NSCA_INTERSECT() && (X) && NSCA_NEXT() && (Y) && NSCA_END()

WITHIN Operator

[0122] The syntax for the “within” operation is:

BNF:

```
sequence_expr :=
..
sequence_expr 'nsc_within' sequence_expr
..
```

This expression can be specified with a macro:

[0123] NSC_WITHIN (X, Y)

This expression is translated as:

[0124] NSCA_WITHIN() && (X) && NSCA_NEXT()
&& (Y) && NSCA_END()

THROUGHOUT Operator

[0125] The syntax for the “throughout” operation is:

BNF:

```
sequence_expr :=
..
systemc_simple_expression 'nsc_throughout' sequence_expr
..
```

This expression can be specified with a macro:

[0126] NSC_THROUGHOUT(X, Y)

This expression is translated as:

[0127] NSCA_THROUGHOUT() && (X) &&
NSCA_NEXT() && (Y) && NSCA_END()

FIRST MATCH operator

[0128] The syntax for the first match operation is:

BNF:

```
sequence_expr :=
..
'nsc_first_match' '(' sequence_expr { , sequence_match_item } ')'
..
```

This expression can be specified with a macro:

[0129] NSC_FIRST_MATCH(X)

The sequence match item can be specified with the NSC_MATCH macro connected with the “&&” operator as:

[0130] NSC_FIRST_MATCH(X && NSC_MATCH(Y))

This expression is translated as:

[0131] NSCA_FIRST_MATCH () && (X) &&
NSCA_END()

Sequence Declaration

[0132] A sequence can be described as a sequence declaration (e.g., a function).

```
sequence_declaration :=
'nsc_sequence' '(' [ [ formal_args ] ')' ] '{'
local_variable_decl
sequence_expr
'}
```

[0133] A sequence declaration is translated as an integer type function. Two macros NSCFUNC and NSCFUNCEND are used before and end of the sequence translation.

```
int <sequence_name> ( <arguments> ) {
<variable declaration>
NSCA_SEQ_BEGIN( );
<sequence expression>
NSCA_SEQ_END( );
}
```

Sequence Instantiation

[0134] The sequence instantiation is used to call another sequence description.

BNF

```
sequence_expr :=
..
|
sequence_instance
sequence_instance:=
sequence_identifier [ '(' actual_args ')' ]
```

This can be written with a macro as:

[0135] NSC_CALL(<sequence_func_call>)

This expression is translated as:

[0136] NSCA_CALL() && (<property_function_call>)
&& NSCA_END()

Appendix B

[0137] The translated temporal expression is connected with various construct functions to construct the data structure during the parsing phase. APPENDIX B provides the descriptions of such construct functions.

[0138] The NSCA_PROP_BEGIN() function creates a new nsca_assert_expr structure. It keeps the return address to ret_addr. The NSCA_PROP_BEGIN() function calls the “nsca_top.set_expr_to_body()” function to attach the expression to the body. If this property is not the top of the temporal expression, then the function creates the local stack copy for repetitive execution. The function then returns 1 to continue the evaluation.

```

int NSCA_PROP_BEGIN() {
    nsca_assert_expr *exp = new nsca_assert_expr
    ( nsca_t_property );
    exp->set_ret_addr( nsc_asm_get_ret_addr( ) );
    nsca_top.set_expr_to_body( exp );
    if( exp->get_parent( ) // child property
        exp->set_stack( nsc_asm_copy_stack( ) );
    return 1; /* parsing phase */
}

```

[0139] The NSCA_PROP_END() function is called at the end of the property. If the current property is the top of the temporal expression, it invokes the “nsca_top.eval_engine()” function to evaluate the assertion.

```

int NSCA_PROP_END()
/* finish constructing the data structure */
nsca_assert_expr *exp = nsca_top.get_current( );
exp->set_exit_addr( nsc_asm_get_ret_addr( ) );
nsca_top.end_expr( );
if( !exp->get_parent( ) // if this is the top
{
    nsca_top.gen_top_bubble( ); // create the initial bubble
    /* calling the evaluation module 204 */
    ret_value = nsca_top.eval_engine( 0 );
    /* evaluating at the evaluation phase */
    nsc_asm_set_ret_addr( nsca_top.get_ret_addr( ) );
    /* ret_addr is set by evaluation_engine( ) function */
    return ret_value; /* */
}
else return 1; // not top, continue parsing
}

```

[0140] During the parsing phase, the NSCA_SEQ_BEGIN() function creates a new “nsca_assert_expr” structure (type=nsca_t_sequence). The function keeps the return address to ret_addr. If this sequence is not the top of the temporal expression, then it creates the local stack copy for repetitive execution. The function then returns 1 to continue the evaluation. During the evaluation phase, the function calls the evaluation module 204.

```

int NSCA_SEQ_BEGIN() {
    if( parse_phase ) {
        nsca_assert_expr *exp = new nsca_assert_expr
        ( nsca_t_sequence );
        exp->set_ret_addr( nsc_asm_get_ret_addr( ) );
        nsca_top.set_expr_to_body( exp );
        if( exp->get_parent( ) // child property
            exp->set_stack( nsc_asm_copy_stack( ) );
        return 1; /* parsing phase */
    }
    else {
        /* calling the evaluation phase engine
        with argument 1 that tells the evaluation is true */
        int ret_value = nsca_top.eval_engine( 1 );
        /* evaluating b1 at evaluation phase */
        nsc_asm_set_ret_addr( nsca_top.get_return_addr( ) );
        /* ret_addr is set by evaluation_engine( ) function */
        return ret_value; /* */
    }
}

```

[0141] The NSCA_SEQ_END() function is called at the end of the sequence. The function keeps its return address to

exit_addr field of the current sequence structure. If the current sequence is the top of the temporal expression, it invokes the “nsca_top.eval_engine()” function to evaluate the assertion.

```

int NSCA_SEQ_END() {
    /* finish constructing the data structure */
    nsca_assert_expr *exp = nsca_top.get_current( );
    exp->set_exit_addr( nsc_asm_get_ret_addr( ) );
    nsca_top.end_expr( );
    if( !exp->get_parent( ) // if this is the top
    {
        nsca_top.gen_top_bubble( ); // create the initial bubble
        /* calling the evaluation phase engine */
        ret_value = nsca_top.eval_engine( 0 );
        /* evaluating at evaluation phase */
        nsc_asm_set_ret_addr( nsca_top.get_ret_addr( ) );
        /* ret_addr is set by evaluation_engine( ) function */
        return ret_value; /* */
    }
    else return 1;
}

```

[0142] The NSCA_NEXT() function is called in the middle of the two operand operation. The function moves the current expr pointer to the parent, and switches the child pointer usage to the child_b side.

```

int NSCA_NEXT() {
    /* finish constructing the data structure */
    nsca_assert_expr *exp = nsca_top.get_current( );
    exp->set_exit_addr( nsc_asm_get_ret_addr( ) );
    nsca_top.close_expr( );
    exp = nsca_top.get_current( );
    exp->switch_child( );
    return 1;
}

```

[0143] The NSCA_END() function is called at the end of a primitive sequence. The function keeps the return address as the exit address of the current expr, and closes the current expr.

```

int NSCA_END() {
    nsca_assert_expr *exp = nsca_top.get_current( );
    exp->set_exit_addr( nsc_asm_get_ret_addr( ) );
    nsca_top.close_expr( );
}

```

[0144] The NSCA_BOOL() function constructs a new expr structure with type=nsca_t_bool. The function saves the return address to ret_addr field of the expr. The function returns 0 to avoid the actual execution of the Boolean section.

```

int NSCA_BOOL() {
    nsca_assert_expr *exp = new nsca_assert_expr ( nsca_t_bool );
    exp->set_ret_addr( nsc_asm_get_ret_addr( ) );
    nsca_top.set_expr_to_body( exp );
    return 0; /* parsing phase */
}

```

[0145] The NSCA_NOT() function constructs a new expr structure with type=nsca_t_not. The function saves the return address to ret_addr field of the expr.

```
int NSCA_NOT() {
    nsca_assert_expr *exp = new nsca_assert_expr ( nsca_t_not );
    exp->set_ret_addr( nsc_asm_get_ret_addr( ) );
    nsca_top.set_expr_to_body( exp );
    return 1;
}
```

[0146] The following NSCA_PAND() to NSCA_FIRST_MATCH functions have similar code as the NSCA_NOT() function, but with different type field.

[0147] The NSCA_PAND() function constructs a new expr structure with type=nsca_t_pand.

[0148] The NSCA_POR() function constructs a new expr structure with type=nsca_t_por.

[0149] The NSCA_AND() function constructs a new expr structure with type=nsca_t_and.

[0150] The NSCA_OR() function constructs a new expr structure with type=nsca_t_or.

[0151] The NSCA_IMP() function constructs a new expr structure with type=nsca_t_imp.

[0152] The NSCA_NOIMP() function constructs a new expr structure with type=nsca_t_noimp.

[0153] The NSCA_CALL() function constructs a new expr structure with type=nsca_t_call. The function keeps the return address to the structure.

[0154] The NSCA_CALL_END() function returns 1 during the parsing phase. The function calls the evaluation module 204 during the evaluation phase.

```
NSCA_CALL_END() {
    if ( parse_pase ) return 1;
    else {
        /* calling the evaluation phase engine
           with argument 1 that tells the evaluation is true */
        int ret_value = nsca_top.eval_engine( 1 );
        /* evaluating b1 at evaluation phase */
        nsc_asm_set_ret_addr ( nsca_top.get_return_addr( ) );
        /* ret_addr is set by evaluation_engine( ) function */
        return ret_value ; /* */
    }
}
```

[0155] The NSCA_SEQUENCE() function is called at the beginning of the sequence expression. The function constructs a new expr structure with type=nsca_t_sequence.

[0156] The NSCA_INTERSECT() function constructs a new expr structure with type=nsca_t_intersect.

[0157] The NSCA_WITHIN() function constructs a new expr structure with type=nsca_t_within.

[0158] The NSCA_THROUGHOUT() function constructs a new expr structure with type=nsca_t_throughout.

[0159] The NSCA_FIRST_MATCHED() function constructs a new expr structure with type=nsca_t_first_match.

[0160] The NSCA_DLY() function constructs a new expr structure with type=nsca_t_dly. The function sets m and n values to start_count and end_count fields.

```
int NSCA_DLY(int m, int n) {
    nsca_assert_expr *exp = new nsca_assert_expr ( nsca_t_dly );
    exp->set_ret_addr( nsc_asm_get_ret_addr( ) );
    exp->set_start_count( m );
    exp->set_end_count( n );
    nsca_top.set_expr_to_body( exp );
    return 1;
}
```

[0161] The following NSCA_CREP() to NSCA_FIRST_GOTO functions have similar code as the NSCA_DLY() function, but with different type field.

[0162] The NSCA_CREP(int m, int n) function constructs a new expr structure with type=nsca_t_crep. The function sets m and n values to start_count and end_count fields.

[0163] The NSCA_NREP(int m, int n) function constructs a new expr structure with type=nsca_t_nrep. The function sets m and n values to start_count and end_count fields.

[0164] The NSCA_GOTO(int m, int n) function constructs a new expr structure with type=nsca_t_goto. The function set m and n values to start_count and end_count fields.

[0165] The NSCA_TRUE() function and the NSCA_FALSE function are called at the evaluation phase. The NSCA_TRUE() function is used to capture the Boolean expression that ends with the true result. The NSCA_FALSE() function is used to capture the Boolean expression that ends with the false result. The evaluation module 204 evaluates the Boolean expression that is represented as an instance of a structure called "bool_expr." The evaluation module 204 sets the captured return address of the NSCA_BOOL() function to the "ret_addr" variable and returns 1. The lower NSCA functions set this "ret_addr" variable as the return address and return with the value passed from the evaluation module 204. In response to the evaluation module 204 completing the evaluation of the Boolean expression, the NSCA_TRUE() or NSCA_FALSE() function captures the result. These functions pass the result as an argument and call the evaluation module 204 again.

Appendix C

[0166] APPENDIX C provides four assembler functions used to execute a temporal expression.

[0167] The nsc_asm_get_ret_addr() function reads and calls the stack frame of the current function and returns the return address of the function.

[0168] The nsc_asm_set_ret_addr() function takes an address as the argument and sets it to the return address location of the current call stack frame.

[0169] The nsc_asm_copy_stack() function copies the local stack frame between the base pointer and the current stack pointer to the allocated memory table and returns the table address.

[0170] The nsc_set_stack() function takes the table address and set it to the stack frame.

Appendix D

[0171] APPENDIX D provides a description of the NFA mechanism for evaluating a temporal expression. The actual evaluation of the temporal expression is executed with the `nsca_top.eval_engine( )` function. The evaluation is done with an NFA mechanism. Each active state over the temporal structure is represented with the “`nsca_nfa_bubble`” structure, as shown below.

```
class nsca_nfa_bubble {
..
private:
nsca_nfa_state state ; // current state
nsca_nsa_state sub_state ; // additional state info.
int count ; // generic counter
int time_a ; // generic time reg a
int time_b ; // generic time reg b
nsca_assert_expr *expr ; // pointer to the structure
nsca_nfa_bubble *parent ; // pointer to parent bubble
nsca_nfa_bubble *child ; // pointer to child bubble
nsca_nfa_bubble *child_b ; // another child bubble
nsca_nfa_queue *queue ; // pointer to the queue top
nsca_nfa_bubble *prev ; // prev ptr in queue
nsca_nfa_bubble *next ; // next ptr in queue
nsca_nfa_bubble *fcall ; // call pointer for instance
stack_table stack_frame ; // local var copy for call
}
```

[0172] The bubble has the fields “state” and “sub_state” to represent the status.

[0173] Table 1 shows the status field condition.

TABLE 1

state name	meaning
<code>nsca_s_active</code>	active state
<code>nsca_s_delay</code>	delay state (cycle delay)
<code>nsca_s_wait</code>	waiting for child evaluation
<code>nsca_s_zombie</code>	dead bubble waiting for clean up
<code>nsca_sub_idle</code>	Idle sub state
<code>nsca_sub_s0</code>	s0 sub state
<code>nsca_sub_s1</code>	s1 sub state

[0174] Depending on the state, each bubble is placed in different queues (e.g., a linked list). The “`nsca_active_queue`” queue holds the active bubbles. The “`nsca_delay_queue`” queue holds the bubbles that wait for a cycle delay. The “`nsca_wait_queue`” queue holds the bubbles that wait for child evaluation. The “`nsca_zombie_queue`” queue holds the dead bubbles that are being cleaned. The bubbles are connected as double-linked list using previous and next pointers. The queue field in the bubble structure points to the queue that it belongs. If a bubble is placed in the delay queue, it is given the waiting count to the count field. It is then placed in a proper location in the queue such that the queue conforms to a linked list of decrement counter. The count value is adjusted such that the actual wait count is the sum of the count values of the bubbles located before the corresponding bubble. For example:

```
delay_queue->A(count=5)->B(count=3)->C (count=2)->..
```

The actual wait cycle of the bubble C is 5+3+2=10. With this structure, the count decrement can be done by decrementing the count in the top of the queue.

[0175] The “eval_engine” function handles the evaluation of the temporal logic.

[0176] The pseudo code for the function is shown below:

```
int eval_engine ( int eval_result ) {
if( eval_bubble ) {
// active eval bubble exist
eval_bubble.done( eval_result ) ;
}
if( eval_bubble ) {
// eval_bubble must be cleared
// if there's no evaluation needed
// reevaluate the boolean/sequence match item
return 1 ;
}
while( 1 ) { // forever loop
// evaluation loop
while( top = nsca_active_queue.dequeue( ) ) {
top.eval( ) ; // evaluate the bubble
if( eval_bubble ) {
// reevaluate the boolean/sequence match item
return 1 ;
}
}
if( top = nsca_delay_queue.dequeue( ) ) {
wait( top.get_count( ) ) ; // cycle wait
// wake up from cycle wait
while(
nsca_delay_queue( ).get_top( ).get_count( ) == 0
) {
// if other bubble in the queue has count
// value 0,
// then move them to the active queue
top = nsca_delay_queue.dequeue( ) ;
nsca_active_queue.enqueue( top ) ;
}
while( top = nsca_zombie_queue.dequeue( ) ) {
delete top ; // clean up the dead bubble
}
if( eval_done( ) ) {
return 1 ; // exit from the eval loop
}
}
}
```

[0177] Each active bubble is evaluated from the `eval_engine( )` function. The bubble class provides the evaluation functions as the following:

[0178] The `evalo` function is called if the bubble is placed in the active queue. The action for the evaluation is determined by the sequence structure where it is located at.

```
eval( ) {
switch( expr->type ) { // action per expr type
case ..
}
}
```

[0179] The `done( )` function is called at the completion of Boolean or sequence match item evaluation.

```
done( int eval_result ) {
eval_bubble = 0 ; // clear active bubble
if( expr->type == nsca_t_bool ) {
```

-continued

```

if( eval_result ) {
    if( expr->next ) {
        expr = expr->next ;
        nsc_active_queue.enqueue( this ) ;
    }
    else {
        if( parent ) parent.matched( this ) ;
        else nsc_top.eval_done( 1 ) ;
    }
}
else {
    if( parent ) parent.unmatched( this ) ;
    else nsc_top.eval_done( 0 ) ;
}
}
else { // nsca_t_match
    if( expr->next ) expr = expr->next ;
    else {
        if( parent ) parent.matched( this ) ;
        else nsc_top.eval_done( 1 ) ;
    }
}
}
}

```

[0180] The matched() function is called if a bubble evaluation reaches to a “match” condition and the bubble informs the condition to its parent bubble.

```

matched( nsca_nfa_bubble *child ) {
    switch( expr->type ) { // ..
        // proper action per type
    }
}

```

[0181] The unmatched() function is called if a bubble evaluation reaches to an “unmatched” condition and the bubble informs the condition to its parent bubble.

```

unmatched( nsca_nfa_bubble *child ) {
    switch( expr->type ) { // ..
        // proper action per type
    }
}

```

[0182] Depending on the sequence expr where the active bubble is located, the proper evaluation action will be taken via the eval() function. The following text explains the action flow of various expr types.

nsca_t_property

[0183] For the property expression, if the child type is not nsca_t_pand or nsca_t_por, a new bubble for the child is created each cycle and placed to the active queue. The bubble itself is placed in the delay queue for one cycle to be ready for the bubble in the next cycle. If the child is nsca_t_and or nsca_t_por, a single bubble is created for the child and placed in the active queue at the beginning. The bubble itself is placed in the wait queue. In response to receiving the matched() call, the matched() function of the parent is called, if a parent exists. In response to receiving the unmatched() call, the unmatched() function of the parent is called. If this is the top, the nsca_assertion_error() function is reported.

nsca_t_sequence

[0184] For the sequence expression, a new bubble for the child is created and placed to the active queue. The bubble itself is placed in the wait queue. In response to receiving the matched() call, the matched() function of the parent is called, if a parent exist. Otherwise, “1” is returned as the evaluation result, if there is no active child bubble. In response to receiving the unmatched() call, the unmatched() function of the parent is called. If this is the top and there is no active child bubble, “0” is returned as the evaluation result.

nsca_t_imply

[0185] If sub_state is idle, a new bubble for the child is created and placed to the active queue. This bubble itself is placed in the wait queue with sub_state=s0. In response to receiving the matched() call, if sub_state=s0, then the sub_state is changed to s1 (stays in wait queue), and a new bubble for its child_b is created and placed to the active queue. If sub_state=s1, then the matched() function of its parent is called, and the bubbled is placed to the zombie queue. In response to receiving the unmatched() call, if the sub_state=s0, then the bubble places itself to the zombie queue. If sub_state=s1, then the unmatched() function of the parent is called.

nsca_t_noimply

[0186] If sub_state is idle, a new bubble for the child is created and placed to the active queue. This bubble itself is placed in the wait queue with sub_state=s0. In response to receiving the matched() call, if sub_state=s0, then the sub_state is changed to s1 and placed in the delay queue for one cycle. In response to exiting from the delay queue, the bubble moves to the wait queue again, and a new bubble for its child_b is created and placed to the active queue. If sub_state=s1, then the matched() function of the parent is called, and the bubble places itself to the zombie queue. In response to receiving the unmatched() call, if sub_state=s0, then the bubble places itself to the zombie queue. If sub_state=s1, then the unmatched() function of the parent is called.

nsca_t_not

[0187] A new bubble for the child is created and placed to the active queue. This bubble itself is placed in the wait queue with sub_state=s0. In response to receiving the matched() call, the unmatched() function of the parent is called. In response to receiving the unmatched() call, the matched() function of the parent is called.

nsca_t_pand

[0188] Two new bubbles for the children are created and placed to the active queue each cycle. The bubble itself is placed in the wait queue to wait for child completion with sub_state=s0. In response to receiving the matched() call, the condition that the two child properties are both matched is detected, and the parent’s matched() function is called. In response to receiving the unmatched() call, the parent’s unmatched() function is called.

nsca_t_por

[0189] Two new bubbles for the children are created and placed to the active queue each cycle. The bubble itself is placed in the wait queue to wait for child completion with

sub_state=s0. In response to receiving the matched() call, the parent's matched() function is called. In response to receiving the unmatched() call, if sub_state=s0, the sub_state changes to s1 and stays in the wait queue. If sub_state=s1, the parent's unmatched() function is called.

nsca_t_and

[0190] Two new bubbles for the children are created and placed to the active queue. The bubble itself is placed in the wait queue to wait for child completion with sub_state=s0. In response to receiving the matched() call, the condition that the two child sequences are both matched is detected, and the parent's matched() function is called. In response to receiving the unmatched() call, the parent's unmatched() function is called.

nsca_t_or

[0191] Two new bubbles for the children are created and placed to the active queue. The bubble itself is placed in the wait queue to wait for child completion with sub_state=s0. In response to receiving the matched() call, the parent's matched() function is called. In response to receiving the unmatched() call, if sub_state=s0, the sub_state changes to s1 and stays in the wait queue, if sub_state=s1, the parent's unmatched() function is called.

nsca_t_intersect

[0192] Two new bubbles for the children are created and placed to the active queue. The bubble itself is placed in the wait queue to wait for child completion with sub_state=s0. In response to receiving the matched() call, and if sub_state=s0, the sub_state changes to s1 and moves to the ready queue. If the bubble is reached to be executed, and if the state is still s1, the sub_state changes to s0 and moves back in the wait queue. If sub_state=s1, then the parent's matched() function is called, and the sub_state is changed to s2. In response to receiving the unmatched() call, the parent's unmatched() function is called.

nsca_t_within

[0193] Two new bubbles for the children are created and placed to the active queue. The bubble itself is placed in the wait queue to wait for child completion with sub_state=s0. In response to receiving the matched() call, it is checked if the first sequence is within the second sequence. Otherwise, the parent's unmatched() function is called. In response to receiving the unmatched() call, the parent's unmatched() function is called.

nsca_t_throughout

[0194] A new bubble for the children sequence is created and placed to the active queue. Until the matched() item is received from the sequence, a new bubble for evaluating the Boolean expression is created each cycle. In response to receiving the unmatched() call from the first Boolean bubble, the parent's unmatched() function is called. In response to receiving the matched() call from the second sequence, the parent's matched() function is called. In response to receiving the unmatched() call from the second sequence, the parent's unmatched() function is called.

nsca_t_first match

[0195] A new bubble for the child is created and placed to the active queue. The bubble itself is placed in the wait

queue to wait for the child. In response to receiving the matched() call, the child bubble is cleaned up and the parent's matched() function is called. In response to receiving the unmatched() call, the parent's unmatched() function is called.

nsca_t_dly

[0196] The bubble is placed in the delay queue for the initial delay value (start_count). Then a new bubble for the child is created and placed to the active queue each cycle till the end_count is reached. Then the bubble itself is placed in the wait queue to wait for children completion. In response to receiving the matched() call, the parent's matched() function is called. In response to receiving the unmatched() call, and no child bubble is left, the parent's unmatched() function is called.

nsca_t_crep

[0197] A new bubbles for the child is created and placed to the active queue. In response to receiving the matched() call (or if "zero" match is the condition), a new bubble is created for the child till the end_count is reached. The counter is counted up, and the condition is checked for match. If the count is larger than the start_count, the matched() function of the parent is called. In response to receiving the unmatched() call, the parent's unmatched() function is called.

nsca_t_goto

[0198] A new bubble for the child is created and placed to the active queue each cycle till the match count reaches the end_count. In response to receiving the matched() call (or if "zero" match is the condition), the counter is counted up. If the condition matched, the parent's matched() function is called. If not consecutive, the parent's unmatched() function is called.

nsca_t_nrep

[0199] A new bubble for the child is created and placed to the active queue each cycle till the first match is detected. In response to receiving the first matched() call (or if "zero" match is the condition), the parent's matched() function is called each cycle.

nsca_t_call

[0200] A new bubble for the child is created and placed to the active queue. The pointer to this bubble is copied to the fcall pointer of the child, and the pointer value inherits the bubbles under the child (until a new call is encountered). The bubble itself is placed in the wait queue. In response to evaluating the Boolean expression or sequence match item, the fcall pointer is checked. If the fcall pointer is not null, the function call sequence is processed by returning before the function call and capturing at the NSCA_SEQ_BEGIN() call. In response to capturing at the sequence begin, the local variable is copied to the stack to continue the Boolean evaluation. In response to completing the Boolean or sequence match item evaluation, the local variable is copied and returned at the exit address. The exit is recaptured at the NSCA_SEQ_END() function. If the call is nested, multiple call sequences are executed. The bubble is purged if the parent is cleaning up the children bubbles.

nsca_t_bool

[0201] The Boolean expression is evaluated by setting the return address and returning “1.” The result is returned with reactivation. The sub-status is used to handle the condition, before the evaluation and after the evaluation. If the result is “true,” then the parent’s matched() function is called. Otherwise, the parent’s unmatched() function is called.

nsca_t_match

[0202] The sequence match item expression is evaluated by setting the return address and returning with “1.” The result is returned with reactivation. The sub-status is used to handle the condition, before the evaluation and after the evaluation. The parent’s matched() function is called.

Appendix E

[0203] During the parsing phase, the temporal expression is parsed and the data structure that represents the temporal expression is dynamically constructed. The temporal data structure is created under a thread structure. There is one active temporal expression structure per single thread. The temporal data structure per thread is hold within a class called “nsca_assertion_top.” The instance “nsca_top” provides various service calls to let the construction and evaluation functions do the proper tasks. This APPENDIX E provides the functions provided by “nsca_top.”

[0204] The “set_expr_to_body(nsca_assert_exp*exp)” function sets the given assert to the body and sets this exp as the current expression. If there is no current expr, the new exp is connected as the current active expr for the thread. If the current expr is open, the new one is connected to its child. There are two child pointers, and which one is used depends on whether NSCA_NEXT() is called or not. If the current expr is not open, the new one is connected to its next.

[0205] The “end_expr()” function closes the current expression. If the current expression is not open, it moves the current expr to its parent before closing it.

[0206] The “close_expr()” function closes the current expression and moves the current expr to its parent.

[0207] The “get_current()” function returns a pointer to the current active expr structure.

[0208] The “eval_engine(int value)” function invokes the evaluation module 204.

[0209] The “set_return_addr(return_address ret)” function sets the return address for future use.

[0210] The “get_return_addr()” function reads out the return address set by the “set_return_addr” function to actually returning to the address.

[0211] The “gen_top_bubble()” function creates the top bubble for evaluation.

[0212] The temporal structure uses a common data structure “nsca_assert_expr” to represent various types of nodes indicating the temporal property. The type field in the “nsca_assert_expr” data structure shows which kind of information a node holds. Table 2 shows the type information.

TABLE 2

type name	Meaning
nsca_t_property	property top
nsca_t_not	not property
nsca_t_pand	And property
nsca_t_por	or property
nsca_t_and	And sequence
nsca_t_or	or sequence
nsca_t_imply	imply (->) property
nsca_t_noimply	no overlapped imply (=>) property
nsca_t_call	sequence instance call
nsca_t_sequence	sequence top
nsca_t_dly	delay operation (@[m, n])
nsca_t_bool	Boolean expression
nsca_t_match	sequence match item
nsca_t_intersect	intersect sequence operation
nsca_t_within	within sequence operation
nsca_t_throughout	throughout sequence operation
nsca_t_first_match	First match sequence operation
nsca_t_crep	consecutive repetition
nsca_t_goto	Goto repetition
nsca_t_nrep	Non-consecutive repetition

[0213] The member fields in the temporal_expr hold the at least one of the following information shown in Table 3:

TABLE 3

Field	meaning
nsca_assert_type	Type field shown in Table 2
int start_count	start count for ranged operation
int end_count	End count for ranged operation
nsca_assertion_expr *parent	pointer to the parent structure in the structure hierarchy. If the structure is at the top, ‘0’ (zero) will be stored.
nsca_assertion_expr *next	pointer to the next structure in the same hierarchy. If the structure is at the end in the hierarchy, 0 (zero) will be stored.
nsca_assertion_expr *child	pointer to the top of children in the structure hierarchy. If the structure is at the bottom, 0 (zero) will be stored.
nsca_assertion_expr *child_b	For binary operation, there is one more child pointer.
return_address ret_addr	return address
return_address exit_addr	Exit address from the current hierarchy
stack_table stack_frame	address to the copy of the stack frame

We claim:

1. A method of evaluating a temporal description within a general purpose programming language, the method comprising:

translating the temporal description into a temporal expression, the temporal expression including a native expression of the general purpose programming language;

generating a data structure that represents the temporal expression without evaluating the native expression; and

evaluating the data structure to execute the temporal expression including the native expression.

2. The method of claim 1, wherein the temporal description is in the form of an extended syntax or a preprocessor macro.

3. The method of claim 1, wherein the temporal expression further comprises one or more construct functions.

4. The method of claim 1, wherein generating the data structure that represents the temporal expression comprises:

parsing the temporal expression to generate the data structure without evaluating the native expression.

5. The method of claim 1, wherein generating the data structure that represents the temporal expression comprises:

storing a return address in a stack; and

avoiding evaluating the native expression by controlling a return value of a construct function to be false.

6. The method of claim 5, wherein evaluating the data structure to execute the temporal expression comprises:

returning to the return address stored in the stack; and

evaluating the native expression by controlling the return value of the construct function to be true.

7. The method of claim 1, wherein evaluating the data structure to execute the temporal expression comprises:

creating one or more nondeterministic finite automata state structures, each nondeterministic finite automata state structure including data to be evaluated at a clock cycle;

storing the one or more nondeterministic finite automata state structures in a queue; and

retrieving a nondeterministic finite automata state structure from the queue in response to passage of clock cycles to evaluate the retrieved nondeterministic finite automata state structure.

8. The method of claim 1, wherein evaluating the data structure to execute the temporal expression comprises:

evaluating a variable within the native expression from a name scope of the native expression's program location.

9. A system for evaluating a temporal description within a general purpose programming language, the system comprising:

a translation module adapted to translate the temporal description into a temporal expression, the temporal expression including a native expression of the general purpose programming language;

a parsing module adapted to generate a data structure that represents the temporal expression without evaluating the native expression; and

an evaluation module adapted to evaluate the data structure to execute the temporal expression including the native expression.

10. The system of claim 9, wherein the temporal description is in the form of an extended syntax or a preprocessor macro.

11. The system of claim 9, wherein the temporal expression further comprises one or more construct functions.

12. The system of claim 9, wherein the parsing module is adapted to:

parse the temporal expression to generate the data structure without evaluating the native expression.

13. The system of claim 9, wherein the parsing module is adapted to generate the data structure that represents the temporal expression by:

storing a return address in a stack; and

avoiding evaluating the native expression by controlling a return value of a construct function to be false.

14. The system of claim 13, wherein the evaluation module is adapted to evaluate the data structure to execute the temporal expression by:

returning to the return address stored in the stack; and

evaluating the native expression by controlling the return value of the construct function to be true.

15. The system of claim 9, wherein the evaluation module is adapted to evaluate the data structure to execute the temporal expression by:

creating one or more nondeterministic finite automata state structures, each nondeterministic finite automata state structure including data to be evaluated at a clock cycle;

storing the one or more nondeterministic finite automata state structures in a queue; and

retrieving a nondeterministic finite automata state structure from the queue in response to passage of clock cycles to evaluate the retrieved nondeterministic finite automata state structure.

16. The system of claim 9, wherein the evaluation module is adapted to:

evaluate a variable within the native expression from a name scope of the native expression's program location.

17. A computer program product having a computer-readable medium having embodied thereon program code for evaluating a temporal description within a general purpose programming language, the program code comprising:

a translation module adapted to translate the temporal description into a temporal expression, the temporal expression including a native expression of the general purpose programming language;

a parsing module adapted to generate a data structure that represents the temporal expression without evaluating the native expression; and

an evaluation module adapted to evaluate the data structure to execute the temporal expression including the native expression.

18. The computer program product of claim 17, wherein the temporal description is in the form of an extended syntax or a preprocessor macro.

19. The computer program product of claim 17, wherein the temporal expression further comprises one or more construct functions.

20. The computer program product of claim 17, wherein the parsing module is adapted to:

parse the temporal expression to generate the data structure without evaluating the native expression.

21. The computer program product of claim 17, wherein the parsing module is adapted to generate the data structure that represents the temporal expression by:

storing a return address in a stack; and

avoiding evaluating the native expression by controlling a return value of a construct function to be false.

22. The computer program product of claim 21, wherein the evaluation module is adapted to evaluate the data structure to execute the temporal expression by:

returning to the return address stored in the stack; and

evaluating the native expression by controlling the return value of the construct function to be true.

23. The computer program product of claim 17, wherein the evaluation module is adapted to evaluate the data structure to execute the temporal expression by:

creating one or more nondeterministic finite automata state structures, each nondeterministic finite automata state structure including data to be evaluated at a clock cycle;

storing the one or more nondeterministic finite automata state structures in a queue; and

retrieving a nondeterministic finite automata state structure from the queue in response to passage of clock cycles to evaluate the retrieved nondeterministic finite automata state structure.

24. The computer program product of claim 17, wherein the evaluation module is adapted to:

evaluate a variable within the native expression from a name scope of the native expression's program location.

* * * * *