

**“Politehnica” University of Bucharest  
Computer Science Engineering Department**

**The Art of Translating Sugar to  
an Automata Language**  
*-Dissertation Thesis-*

Scientific Coordinators:

Prof. Dr. Eng. Irina Athanasiu

Graduate:

Eng. Valentin Stefan Gheorghita

**July 2003**

## **Abstract**

The Formal Verification Technical Committee of Accellera has selected Sugar 2.0 as the basis for the Accellera Standard formal property specification language. Many EDA vendors are beginning to implement the support for this language in their verification tools.

Considering the similitude between the verification languages, a tool which automatically converts from Sugar to all of them is desired. It does not need to be aware of the target language syntax, concentrating only on the common features of the verification languages. The interesting common aspect of these languages is their automata support, which can be exploited by the translation.

# Table of Contents

<b>TABLE OF CONTENTS .....</b>	<b>3</b>
<b>INTRODUCTION .....</b>	<b>4</b>
<b>CHAPTER 1 : AUTOMATA MODEL .....</b>	<b>5</b>
1.1 OVERVIEW .....	5
1.2 PROPOSITIONS.....	6
1.3 CLOCK .....	6
1.4 STATES .....	7
1.5 COUNTERS .....	7
1.6 TRANSITIONS .....	8
1.7 INITIALIZATION.....	10
1.8 FINAL STATES.....	10
1.9 UNFAIR STATES .....	10
<b>CHAPTER 2 : SUGAR TRANSLATION .....</b>	<b>12</b>
2.1 SUGAR 2.0 OVERVIEW .....	12
2.2 TRANSLATION APPROACH .....	13
2.3 VERIFICATION DIRECTIVES AUTOMATA SEMANTICS .....	15
2.4 TEMPORAL EXPRESSION AUTOMATA SEMANTICS.....	16
2.5 SERES AUTOMATA SEMANTICS .....	17
2.6 BOOLEAN AUTOMATA .....	17
<b>CHAPTER 3 : RESULTS – A COMPILER .....</b>	<b>19</b>
3.1 COMPILER ARCHITECTURE .....	19
3.2 PREPROCESSORS .....	20
3.3 THE PARSER .....	20
3.4 VALIDATION ENGINE .....	21
3.5 REDUCTION ENGINE .....	21
3.6 CONVERSION ENGINE .....	21
3.7 OPTIMIZATION ENGINE .....	23
3.8 AUTOMATA WRITER.....	25
<b>CHAPTER 4 : CONCLUSION .....</b>	<b>26</b>
<b>CHAPTER 5 : FUTURE WORK.....</b>	<b>27</b>
<b>APPENDIX A: SUGAR TRANSLATION EXAMPLES.....</b>	<b>28</b>
<b>APPENDIX B: SUGAR SUPPORTED GRAMMAR SUBSET.....</b>	<b>31</b>
<b>APPENDIX C: SUGAR PREPROCESSOR MACROS .....</b>	<b>33</b>
<b>REFERENCES .....</b>	<b>36</b>

## Introduction

The Formal Verification Technical Committee of Accellera has selected Sugar 2.0 as the basis for the Accellera Standard formal property specification language [1]. The definition is now being converted into IEEE-standard Language Reference Manual format. In parallel, many vendors are beginning to implement the language.

As usual with new Electronic Design Automation (EDA) language standards, it is likely that each EDA vendor planning to support Sugar 2.0 will choose to implement some subset of the language initially, and then incrementally grow the subset they support over time. While this approach is natural, it can lead to interoperability issues if different vendors implement very different initial subsets. In order to promote early interoperability among tools that will support this standard, it is desirable to identify the commonality among various vendors' initial implementation plans, and to publish interoperability guidelines reflecting this commonality. The chosen grammar and the list of restriction, obtained after the discussions with *IBM*, *Cadence*, *Verplec*, *@HDL*, *TransEDA* and *Oin* has been published in [2]. The guidelines focus on the Verilog flavor of Sugar.

Currently, all verification tools support a proprietary language for boolean, verification, modeling and temporal layers combined with hardware description languages (Verilog, VHDL). All of these property languages have support for automata. Considering this, the current paper provides a very quick way for converting the selected grammar presented in [2] into nondeterministic automata. Only nondeterministic automata are considered since multiple verification tools support them and also algorithms for their determining exist.

First section of the paper contains the definition of the automata structure used as the translation target. This structure is approached only theoretical, without referring to a real EDA language. Second section presents a study of translation of Sugar 2.0 boolean, temporal and verification layer to the presented automata. The accent is on the semantics of the resulted automata. The result of the research, consisting in a compiler that implements the presented theoretical approach, is presented in the lasts sections of the paper.

# Chapter 1 : Automata Model

## 1.1 Overview

The automata model presented in this section aims to offer a backend language for a compiler which converts Sugar into a proprietary property specification language. The automata are used for properties, behaviors and constraints that need to be checked on the design. An automaton checks the trace and in case of acceptance it may have different behavior, like generate an error or emit an event.

Automata are interpreted over propositions, defined as boolean expressions over HDL signals and events. The events are called *triggers*. The propositions characterize the state of the system at a given point in time. The propositions defined over events fulfill the role of identifying those points in time at which the automaton may change its internal state. The automaton changes its internal state only at the points in time when a trigger occurs. If multiple triggers occur at the same time, the automaton will update its state only once, and the update will reflect the presence of all simultaneous triggers. Propositions which are not defined as events do not affect timing. So, if a proposition depends on an HDL signal, then a change in the value of that signal will not cause the automaton to update its state. If the change persists till the next occurrence of a trigger, the automaton will see the new value and update its state accordingly.

An automaton may emit events upon acceptance and in doing so, it can affect other automata.

Automata may be *non-deterministic*, which means that more than one transition may be enabled at a given point in time. Non-deterministic automata can be described more compactly than deterministic versions. Most of verification tools support non-deterministic automata, determining them during compilation or deal with the non-determinism at run-time by computing all of its runs.

Automata may incorporate one or more *down-counters*. Such a counter can be updated by either initializing it or decrementing it; associated with each counter is a proposition

corresponding to the counter's value being zero. These propositions can be used to predicate state transitions of the automaton. Counters are very helpful for checking properties that involve repetition.

Formally, an automaton is a 7-tuple  $(P, Q, C, R, T_0, FIN, UNF)$  where:

- $P$  is a set of propositions.
- $Q$  is a set of states.
- $C$  a set of counters.
- $R$  is the transitions relation.
- $T_0$  is the set of initializing transitions.
- $FIN$  is the set of final states
- $UNF$  is the set of unfair states.

The following sections describe each automaton element and the relation between them.

## 1.2 Propositions

An automaton defines a possibly empty set of propositions  $P$ , over which that automaton is interpreted. A proposition is associated with an event or a boolean expression over HDL signals. The expression which defines a proposition needs to be valid only at those points in time for which the proposition is needed. The inputs of an automaton that does not have any counters are its propositions. Only transition predicates may refer to proposition.

The expression defining a proposition needs to be evaluated (and therefore it needs to be valid) if the automaton is in a state that has a transition whose predicate refers to the proposition and for the set of events at the current time, that transition may be enabled, depending on the value of the proposition.

## 1.3 Clock

An automaton may define a clock. The clock declaration designates an event as clock of the automaton. Declaring a clock has the following effect:

1. Every transition is implicitly predicated on the presence of the clock event.

2. Every state has an implicit self-transition that is enabled if the clock event does not occur.

So, the state of the automaton may change only at those points in time for which the clock event is present, regardless of any event over a proposition is declared in the automaton.

## 1.4 States

An automaton defines a set of states  $Q$ . Each state has associated a list with its outgoing transitions. Optionally, a state may be designated as a *final state* (see section 1.8), an *normal state*, or as an *unfair state* (see section 1.9).

*Final* states are allowed to have no outgoing transitions; states that are not final states and have no outgoing transitions are redundant, and therefore they are not allowed.

A default final state, named *EPSILON*, or a default stuck state, named *STUCK*, may appear as destination state of a transition. *EPSILON* does not have any outgoing transitions. *STUCK* is not a final state and it does not have any outgoing transitions.

## 1.5 Counters

An automaton defines a possibly empty set of counters  $C$ . A counter provides extra storage in the form of an integer variable. As such, the complete state of an automaton consists of a selection of an element from  $Q$  and an assignment of an integer value to each of its counters. The first part of the state is named the *main state* and to the second part is named the *auxiliary state*. An obvious difference between the main state and the auxiliary state is that  $Q$  is a finite set, whereas the domain of the values of the counters is in principle infinite. Also, the automaton description enumerates all successors of the main state for each state of  $Q$ . It does not enumerate the possible values of the auxiliary state in the next step for each value of the auxiliary state in the current step. Instead, the automaton description gives the function for computing the next value in terms of the current value.

Each counter has a valid expression of type integer, over HDL signals associated with. This expression is the *initializing expression* of the counter. The values of the initializing

expression of each counter, together with the values of the propositions constitute the input to the automaton.

Counters are manipulated through *guarded updates*; if the guard evaluates to *false*, then no update is performed, and the transition that carries that guarded update, is not enabled. Consider a counter *count* and its initializing expression *exp*, four types of guarded updates are supported. Table 1 summarizes these four guarded updates, their guards, and the function that gives the next value of the counter in terms of the current value.

Count guard update	Guard	Next value of count if guard is true
init (c)	$exp > 0$	$exp - 1$
!init (c)	$exp = 0$	count
decr (c)	$count > 0$	$c - 1$
!decr (c)	$count = 0$	count

**Table 1 Count Guards**

Count guard update conditional actions are carried by transitions; for a transition to be enabled, the guards of the elements of its count guard updates list have to be *true*.

A counter can not be decremented (with *decr* or *!decr*) before its initialization (with *init* or *!init*).

## 1.6 Transitions

An automaton defines a transition relation, which is the set of state transitions of the automaton. The description enumerates the outgoing transitions of each state. Each transition specifies the source and destination states, a transition predicate which is a boolean expression over propositions and a count guard updates list. A transition is enabled if the all following conditions are satisfied:

1. The automaton is in the source state of the transition.
2. Its predicate is true for the current values of HDL signals and event.
3. The guards of the elements of its count guard updates list are true for the current values of the counters, and the current values of the initializing expressions.

4. If the automaton defines a clock then the transition predicate is implicitly and-ed with that clock; therefore the clock needs to be true for the transition to be enabled.
5. All predicates are implicitly and-ed with ANY<sup>1</sup> event; this is significant only for transition predicates that may be true for an assignment for which none of the events are assigned true. Consequently, the implicit and-ing ensures that the automaton can only change its state if an event occurs.

If a transition is *taken*, the automaton is in the state indicated by the destination state of that transition in the next cycle and its counters are updated according to the count guard updates list of the transition.

Each state has an implicit self-transition with predicate corresponding to the absence of ANY<sup>1</sup> event; this transition makes sure that if no event occurs, the automaton keeps its state. If the verification tool tries to update the automaton only at times for which ANY<sup>1</sup> event occurs, the implicit self-transition can be omitted.

The transition relation is incomplete if there is a state and a truth assignment of the propositions so that none of that state's transitions (including the implicit self-transition) are enabled. If the automaton is in such a state and these input conditions occur, the automaton is said to be *stuck*. Alternatively, we can say that the automaton takes a default transition into STUCK state (see section 1.4).

If the automaton defines a default clock, then every state has an implicit self-transition that is enabled if the clock event is not present.

R, the transition relation, is defined as a set of 4-tuples:

$$R = \left\{ (q_s, q_d, pre, G) \left| \begin{array}{l} q_s \in Q, q_s = \text{transition source}, q_d \in Q, q_d = \text{transition source}, \\ G = \text{count guard updates list}, pre = \text{transition predicate} \end{array} \right. \right\}$$

---

<sup>1</sup> Event which occurs at every point in time at which another event occurs.

## 1.7 Initialization

An automaton may specify multiple *initial states*, and it may specify that some of its counters are to be initialized. Since counters are manipulated by actions on transitions, an initialization takes the form of a transition. These transitions are called *initializing transitions*.

An automaton may define one or more initializing transitions. Each initializing transition identifies an initial state of the automaton. A initializing transition is enabled if the guards of the elements of its count guard updates list are true. An initializing transition does not have a transition predicate. If an initializing transition is taken the automaton is initialized to the destination state of the transition and the counters of the automaton are initialized as specified by that transition.

$T_0$ , the initializing transitions set, is defined as a set of pairs:

$$T_0 = \{(q, G) \mid q \in Q, G \text{ is count guard updates list}\}$$

## 1.8 Final States

An automaton defines a set  $FIN \subset Q$  of final states. The taking of a transition into a final state may generate an error or emit an event. The predefined final state *EPSILON* may appear as destination of a transition.

Safety properties state that *something bad* shall never happen. An automaton which checks such a property watches for the bad behavior and enters a final state when it sees it.

## 1.9 Unfair States

An automaton defines a possible empty set  $UNF \subset Q$  of unfair states. If by the end of the simulation, the automaton ends up in an unfair state, the automaton accepts the trace.

Liveness properties state that *something good* will eventually happen. When such properties are checked in a simulator, the interpretation becomes that *something good* has to happen before the end of simulation. An automaton may be used for checking a liveness property (eventually corresponds to unfair states). That will ensure that if the eventually is not discharged and therefore, the automaton is still in the unfair state at the end of the simulation, the automaton accepts the trace.

## Chapter 2 : Sugar Translation

### 2.1 Sugar 2.0 Overview

Sugar ([1]) is a formal specification language for hardware. A hardware specification written in Sugar can be used by a formal specification tool, such as a model checker, as well as by an informal verification tool, such as a simulator. For model checking, the Sugar specification can *automatically* be translated into one of the standard temporal logics LTL[3] or CTL[4], possibly augmented with auxiliary state machines. For simulation, the Sugar specification can *automatically* be translated into a simulation checker, which may be represented as an automaton presented in Chapter 1.

Sugar consists of four layers. The boolean layer is comprised of boolean expressions. For instance,  $a$  is a boolean expression, having the value true when the signal  $a$  is high, and false when signal  $a$  is low. Sugar interprets the high signal as true and a low signal as false, independent if whether the signal is active-high or active-low. For the remaining of this document, all signals are considered to be active-high, so  $a$  is a boolean expression having the value true when signal  $a$  is asserted, and false otherwise.

The temporal layer consists of temporal properties which describe the relationships between boolean expressions over time. For instance,

$$\textit{always} (req \rightarrow next \textit{ack})$$

is a temporal property expressing the fact that whenever (*always*) signal  $req$  is asserted, then ( $\rightarrow$ ) at the next cycle (*next*), signal  $ack$  is asserted.

The verification layer consists of directives which describe how the temporal properties should be used by the verification tools. For instance,

$$\textit{assert always} (req \rightarrow next \textit{ack})$$

is a verification directive that tells the tools to verify that the property

$$\textit{always} (req \rightarrow next \textit{ack})$$

holds. Other verification directives include an instruction to *assume*, rather to verify, that a particular temporal property holds, or to specify coverage criteria for a simulation tool. The verification layer also provides a means to group Sugar statements into verification units.

Finally, the modeling layer provides a means to model behavior of designs inputs, and to declare and give behavior to auxiliary signals and variables. The modeling layer is also used to give names to properties and other entities from the temporal layer.

Sugar comes in three flavors, corresponding to the hardware description languages Verilog and VHDL, and to EDL, the environment description language. In the Verilog flavor of Sugar, the boolean layer follows Verilog syntax for boolean expressions, and the modeling layer is a subset of the IEEE standard for Verilog [5] augmented with some special Sugar features.

## 2.2 Translation Approach

The verification tools have own ways to model the environment, so for translation, only verification, temporal and boolean layers of Sugar flavor are considered. For these layers, using the reduced Sugar[2] Verilog flavor grammar a dependencies diagram among the Sugar layers was design.

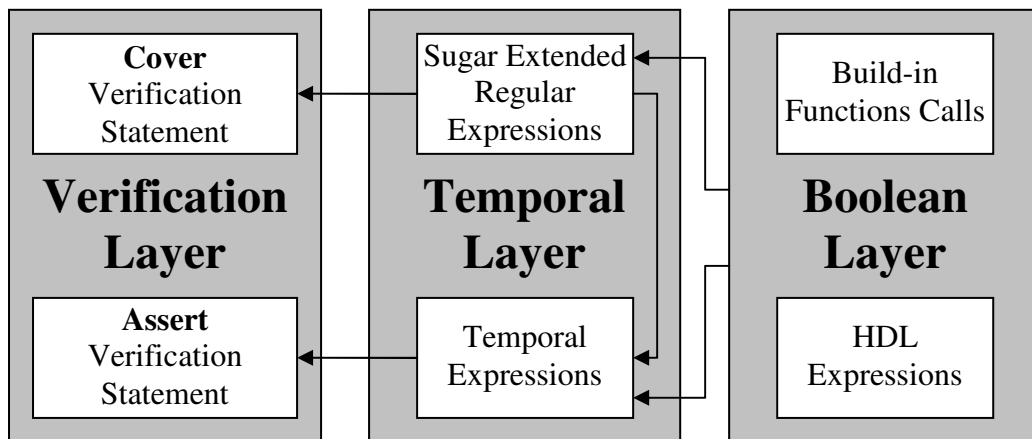


Figure 1 Sugar Layers Dependences Diagram

The boolean layer consists of boolean expressions, in a syntax determined by the flavor of Sugar being used. For instance, the different syntaxes of boolean expressions representing the *bit-wise and* of the most significant four bits of vectors *a* and *b* and the comparison between of the most significant four bits of vector *a* to the bit-string "1111" are presented in Table 2.

Flavor	Bit-wise and syntax	Comparison syntax
Verilog	a[0:3] & b[0:3]	a[0..3]==b1111
VHDL	a(0 to 3) and b(0 to 3)	a(0 to 3) = B"1111"
EDL	a(0..3) & b(0..3)	a(0..3)=1111b

**Table 2 Boolean differences between Sugar flavors**

The temporal layer consists of properties describing the relationships between boolean expressions over time. Sugar does not dictate how time ticks. Rather, it is used to reason about the sequence of signal values as seen by the verification tool being used. It can be used to reason about synchronous designs with a single clock or with multiple clocks, as well as about asynchronous designs; it can be used with cycle-based simulators or event-based simulators, as well as formal verification tools.

The temporal layer is formed by Temporal Expressions (based on temporal operators) and Sugar Extended Regular Expressions (SEREs). The temporal operators can be combined to give quite complicated properties. However, writing such properties is sometimes cumbersome and reading them can be difficult. For instance, the following property:

$$\textit{always} (\textit{reqin} \rightarrow \textit{next} (\textit{ackout} \rightarrow \textit{next} (!\textit{abortin} \rightarrow (\textit{ackin} \ \& \ \textit{next} \ \textit{ackin}))))$$

states that if signal *reqin* is asserted, then if in the next cycle signal *ackout* is asserted, then if in the following cycle signal *abortin* is not asserted, then starting at that cycle, signal *ackin* is asserted for two consecutive cycles. Sugar provides an alternative way to reason about sequences of values which is in many cases more concise and easier to read and write. It is based on an extension of regular expressions, SEREs. As is presented in the layers dependencies diagram, the temporal operators are applied on temporal expressions, SEREs and boolean and SEREs operands for are only SEREs and booleans.

The verification layer provides a way to instruct the verification tools on what to do with the properties via directives to the verification tools. The verification directives are only applied on temporal layer production.

During the translation, every non-terminal from Sugar Grammar [2] is converted into an automaton based on the automata already created for the terminals and non-terminals that form the production. In order to be used, each resulted automaton must follow a specific semantics, depending on the type of production (e.g. SERE, Temporal Expression, Verification Statement). The following sections define the semantics of the resulted automata for each type of production, considering the situations where the productions are used in the Grammar and what it is expected from the Sugar statements.

Using the dependencies among layers, the top-down approach was selected for Sugar grammar analyses. This means that first the production types from verification layer are analyzed, and the semantics of their resulted automata is defined. After this, the ones from the temporal layer, followed by the boolean layer will be analyzed.

## 2.3 Verification Directives Automata Semantics

*Assert* and *cover* are the Sugar verification directives considered for translation. The *assert* directive instructs the verification tool to verify that a property holds. The *cover* directive directs the verification tool to check if a certain path was covered by the verification space based on a simulation test suite or a set of given constraints. The verification directives semantics are presented in Appendix B.

State type	Description
FINAL	Verification directive should fire
STUCK	-- <i>Unused</i> --
NORMAL	Verification directive may fire in the future
UNFAIR	Like a normal state, except on the last simulation cycle, when it becomes final.

**Table 3 Semantics of Verification Directive Resulted Automata**

Every verification directive is converted in an automaton (which is a checker for a verification tool), based on the automaton already generated for the verified temporal formula or SERE. Table 3 describes the semantics of every state type for the resulted

automaton. This semantics was defined based on the action done by the verification directives.

The resulted automaton for a *cover* directive must fire<sup>2</sup> if its SERE accepts the input path. The resulted automaton will emit an event upon acceptance.

The resulted automaton for an *assert* directive fire if input path wasn't accepted by its *temporal expression* and there it is no chance to be accepted in the future. The resulted automaton will generate an error message upon acceptance.

## 2.4 Temporal Expression Automata Semantics

The temporal expressions are used only in the *assert* verification directives. Based on the action done by this directive, the Table 4 presents the semantics of resulted automaton for a temporal expression. The resulted temporal expression automata are obtained by combining the already generated automata for the input operands (SEREs, temporal expressions or boolean expressions). The way in which these input automata are combined depends by the operator. The supported temporal operators and their operands types are showed in [2].

State type	Description
FINAL	The formula does not hold and will not hold until the end of the simulation. (This may be decided before the end of the simulation).
STUCK	-- <i>Unused</i> --
NORMAL	At least one prefix is followed.
UNFAIR	The formula does not hold if this automaton is in this state on the last simulation cycle.

Table 4 Semantics of Temporal Expressions Resulted Automata

---

<sup>2</sup> An automaton fires on a cycle if it is in *FINAL* state, or if the cycle is the last simulation cycle and the automaton is in an *UNFAIR* state.

## 2.5 SEREs Automata Semantics

The SEREs are used by the *cover* verification directives and by temporal expressions. Based on the action done by *cover* directive, the Table 5 presents the semantics of resulted automaton for a SERE.

State type	Description
FINAL	A match was found
STUCK	A prefix was abandoned.
NORMAL	At least one prefix is followed.
UNFAIR	-- <i>Unused</i> --

**Table 5 Semantics of SEREs Resulted Automata**

In some case, when a SERE appears as a non-terminal in temporal expression production, the expected automaton semantics is the temporal expression automaton semantics. Considering this, it is defined an algorithm for converting from SEREs automaton semantics to one similar to temporal expression automaton semantics (Table 6). The algorithm first determines the SERE's non-deterministic automaton, and then computes its complement.

State type	Description
FINAL	All prefixes were abandoned.
STUCK	A match was found.
NORMAL	At least one prefix is followed.
UNFAIR	-- <i>Unused</i> --

**Table 6 SEREs similar semantics to temporal expression**

The SEREs automata are obtained by combining the already generated automata for the input operands (SEREs or boolean expressions). The way in which these input automata are combined depends by the operator. The supported temporal operators and their operands types are showed in [2].

## 2.6 Boolean Automata

The boolean layer consists in HDL boolean expressions and built-in functions. The resulted automata respect the same semantics as for SEREs automata. The Table 7 presents the resulted automata for the boolean terminals from the grammar.

Boolean layer element	Automata (See Appendix A for The Legend)
HDL Expression <code>a</code>	<p>The automata for the HDL expression 'a' consists of three states: State 1 (yellow), State 2 (green), and State 3 (red). State 1 is the start state, indicated by a downward arrow. Transitions are: State 1 to State 2 on input 'a', and State 1 to State 3 on input '!a'.</p>
Build-in function <code>prev(a)</code>	<p>The automata for the build-in function 'prev(a)' consists of four states: State 1 (yellow), State 2 (yellow), State 3 (green), and State 4 (red). State 1 is the start state. Transitions are: State 1 to State 2 on input 'a', State 1 to State 4 on input '!a', and State 2 to State 3 on input 'TRUE'.</p>
Build-in function <code>rose(a)</code>	<p>The automata for the build-in function 'rose(a)' consists of three states: State 1 (yellow), State 2 (green), and State 3 (red). State 1 is the start state. Transitions are: State 1 to State 2 on input 'a &amp; !prev(a)', and State 1 to State 3 on input '!a   prev(a)'.</p>
Build-in function <code>fell(a)</code>	<p>The automata for the build-in function 'fell(a)' consists of three states: State 1 (yellow), State 2 (green), and State 3 (red). State 1 is the start state. Transitions are: State 1 to State 2 on input '!a &amp; prev(a)', and State 1 to State 3 on input 'a   !prev(a)'.</p>

Table 7 The boolean layer resulted Automata

## Chapter 3 : Results – A Compiler

### 3.1 Compiler Architecture

Our theoretical approach was implemented in a tool. It is a *Sugar to an Automata Language (Sugar2AL)* compiler, at which more backends may be added, in order to give the possibility to dump the automata in different languages which has specific syntax for automata support. The tool modular design is depicted in Figure 2.

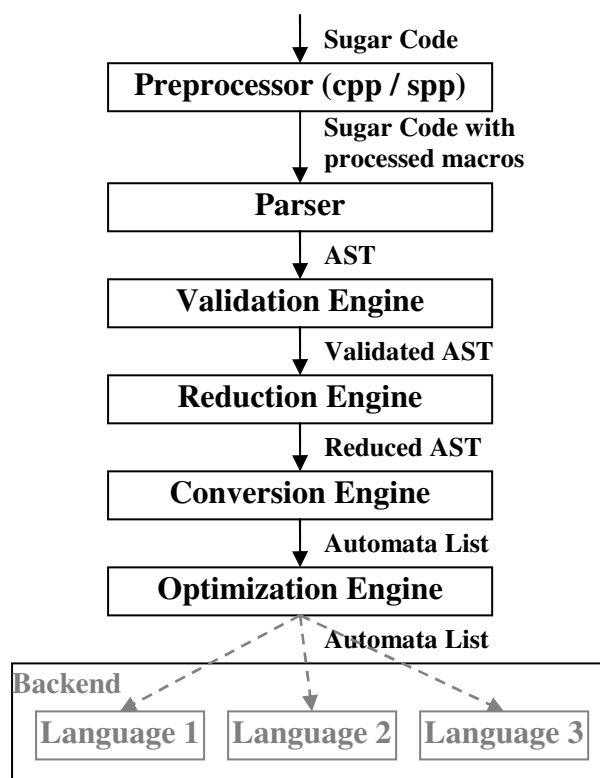


Figure 2 Compiler Architecture

The following sections describe each module of the tool. Some automata, generated using the compiler, are presented in the Appendix A.

## 3.2 Preprocessors

Sugar provides macro-processing capabilities that facilitate the definition of properties. All flavors support *cpp* style pre-processing directives (e.g., `#define`, `#ifdef`, `#else`, `#include`, and `#undef`) and special macros for `%for` and `%if`, which can be used to conditionally or iteratively generate Sugar statements.

The *Sugar2AL* compiler uses *cpp*[6] to handle the *cpp* style pre-processing directives and *spp* (sugar preprocessor) to handle the special macros. *Spp* was developed together with *Sugar2AL* and is based on *vpp*[7]. *Spp* provides an improved set of pre-processor features. New features include `%if <expression>`, `%switch`, `%for`, `%let <var>=<expression>`, and `%while`. The post operators `++` and `--` are supported, not only in the expressions of the extensions, but also in the Sugar code itself. The full support offered by *Spp* is presented in Appendix C.

## 3.3 The Parser

The parser, written in *flex* and *bison*, converts the input sugar files into the corresponding Abstract Syntax Tree (AST). During the conversion the symbols table is filled and the following checks are done:

- Syntactic checks of the grammar: There are constructs that pass parsing but are not in the language (because bison cannot check them). For example, it checks that CTL and syntactic sugar are not combined in formulas.
- No re-declaration of identifiers.
- No unexpected temporal parameters to named properties and named sequences.
- Numeric constants where they are expected. For example, in repetitions such as `[*3]`, `[=3..5]`.
- No calls to undefined functions.
- The arguments list in a function call is consistent with the function definition.

### 3.4 Validation Engine

This module covers all the checks which cannot be done during the AST building. These checks may be general checks or specific checks, depending on the target language (in which the resulted automata are transformed).

### 3.5 Reduction Engine

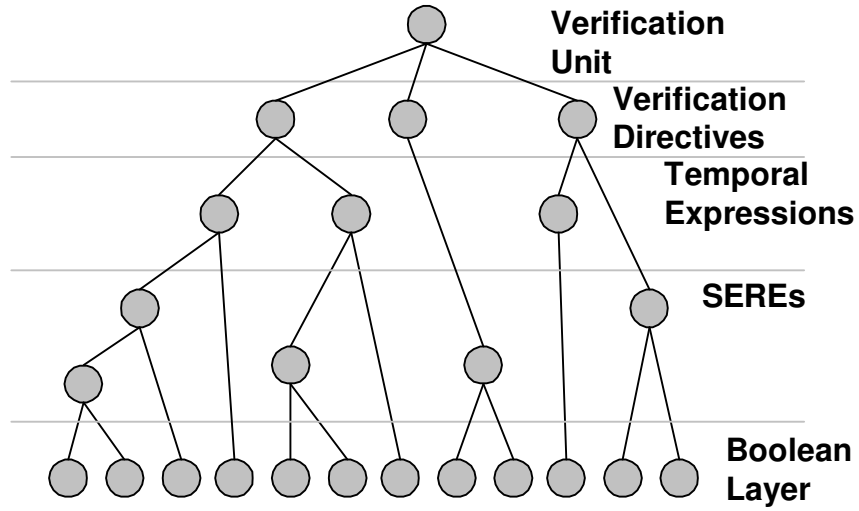
This module purpose is to reduce the number of Sugar operator types used in the AST. The reduction is based on the equivalence of Sugar operators.

The output of the reduction engine is an AST that uses a small number of operator types (named basic operators). The resulted AST may have more nodes than the original one, but the engine which converts it in automata will be simpler to write because it must handle a smaller number of operator types. The rules implemented by this engine are based on [8] and [9]. Some of them are presented bellow:

- `a[+]` = `a;a[*]`
- `a[*i ..]` = `a[*i];a[*]`
- `a[*i .. j]` = `a[*i]; a[* .. j-i]`
- `[*]` = `true[*]`
- `a[=i]` = `{!a[*];a}[i];!a`
- `always a` = `{true [*]} |-> {a}`
- `never a` = `{true [*]; a} |-> {false}`
- `a |=> b` = `{a} |-> {true; b}`

### 3.6 Conversion Engine

This module converts the AST into automata. The resulted automata respect the semantics described in the previous Chapter 2. Considering the Sugar grammar, the input AST is ordered as depicted in Figure 3.



**Figure 3 AST Order Model**

The conversion algorithm traverses the tree in post-order, from the leaves to the root. For every reached node, it builds an automaton, with respect to the generated automata for its children and the information from the node. If the node is a tree leaf (has no children), a simple automaton is generated, as is showed in Section 2.6. The algorithm used for combining children automata depends only on the internal tree node information, which is an operator. An example of a combination algorithm is the one for  $\&\&$  (*sequence length-matching and*) operator:

---

**Input:** An internal tree node that contains the operator  $\&\&$  and has two children. The conversion algorithm takes the children automata as parameters and generates an automaton that implements the operator. Because the  $\&\&$  is applied only on SEREs, the input automata must respect the SEREs automata semantics.

**Input automata:**

$A1 = (P1, Q1, C1, R1, T_01, FIN1, UNF1)$  , first child automaton.

$A2 = (P2, Q2, C2, R2, T_02, FIN2, UNF2)$  , second child automaton.

**Pre-conditions:**  $UNF1 = \emptyset, UNF2 = \emptyset$ , A1 and A2 respect SERE automata semantics.

---

---

**Output:** The resulted production from an  $\&\&$  expression is a SERE. Considering this, the resulted automaton must respect the SEREs automata semantics.

**Output automaton:**  $A_3 = (P_3, Q_3, C_3, R_3, T_0_3, FIN_3, UNF_3)$

**Post-conditions:**  $UNF_3 = \emptyset$ ,  $A_3$  respects SERE automata semantics.

---

**Algorithm:**

$$\begin{aligned} P_3 &= P_1 \cup P_2 \\ Q_3 &= Q_1 \times Q_2 = \{(q_1, q_2) \mid q_1 \in Q_1, q_2 \in Q_2\} \\ C_3 &= C_1 \cup C_2 \\ FIN_3 &= FIN_1 \times FIN_2 = \{(q_1, q_2) \mid q_1 \in FIN_1, q_2 \in FIN_2\} \\ R_3 &= \{((qs_1, qs_2), (qd_1, qd_2), pre_1 \& pre_2, G_1 \cup G_2) \mid \\ &\quad (qs_1, qd_1, pre_1, G_1) \in R_1 \text{ and } (qs_2, qd_2, pre_2, G_2) \in R_2\} \\ T_0_3 &= \{((q_1, q_2), G_1 \cup G_2) \mid (q_1, G_1) \in T_0_1 \text{ and } (q_2, G_2) \in T_0_2\} \end{aligned}$$

---

### 3.7 Optimization Engine

The conversion quality from Sugar to automata affects the resources required by the verification tools. This motivates the search for algorithms that generate efficient automata (e.g. automata with few states, few transitions, and simple predicates). The conversion algorithms try to create the most efficient automaton for every operator, but the automata efficiency is affected by the algorithm generality (for special input cases the algorithm may be modified to generate more efficient automata, but because the number of cases is unlimited, not all of them may be treated). Considering this, on the resulted automata, some optimization algorithms are applied in order to reduce them. In the following paragraphs the current implemented optimization algorithms are presented:

- *Delete Duplicated Propositions:* Identifies all the propositions which refer to the same event or HDL expression and keeps only one of them. All the predicates which use a deleted proposition will use its remaining copy.

- *Delete Duplicated Transitions*: Identifies identical transitions and keeps only one of them. Two transitions  $t_1=(q_{s1}, q_{d1}, pre1, G1)$  and  $t_2=(q_{s2}, q_{d2}, pre2, G2)$  are equals if  $q_{s1}=q_{s2}$ ,  $q_{d1}=q_{d2}$ ,  $G1=G2$  and  $pre1=pre2$ .
- *Merge Parallel Transitions without Count Guard Updates*: Two transitions  $(q_s, q_d, pre1, \emptyset)$  and  $(q_s, q_d, pre2, \emptyset)$  are combined to a single transition  $(q_s, q_d, pre1 || pre2, \emptyset)$ .
- *Delete States with the same Incoming Transitions*: Selects the states with the same type and the same incoming transitions and keeps only one of them. The outgoing transitions of the removed copies are moved to the remaining copy.
- *Delete States with the same Outgoing Transitions*: Selects the states with the same outgoing transitions and keeps only one of them. The incoming transitions of the removed copies are moved to the remaining copy. The type of the remaining state is the strongest state of the selected states (the order of the state type strength, from the smaller to the greatest, is: stuck, normal, unfair and final).
- *Delete Unreachable States*: All the states which are not reachable from at least one initial state, for the all propositions' possible values, are removed together with their incoming and outgoing transitions.
- *Delete States without Path to a Final or Unfair State*: All the states, from which no final or unfair state can be reached for the all propositions' possible values, are removed.
- *Simplify Predicates*: The predicates, which are boolean expressions, are minimized using Quine Mc Cluskey Algorithm (See [10] and [11]).
- *Delete Transitions which have their Predicates Always FALSE*: These transitions are deleted because they are never taken in the automaton simulation.

To obtain smaller conversion time a part of the optimizations may be applied during the conversion stage, to reduce the resulted automata for every AST node. The conversion time is reduced because the combination algorithms are applied on smaller input automata.

### **3.8 Automata Writer**

Different modules may be attached to the compiler in order to dump the internal automata structure to a target language with automata support. Because the paper is not concerned with a target language, no writer module will be presented.

## **Chapter 4 : Conclusions**

We have presented a way to integrate the boolean, temporal and verification layer of Sugar, the language chosen as standard for formal property specification, into a verification tool that uses a proprietary language with automata support. The Sugar modeling layer is not converted and the verification environment must be built in the target language.

The developed tool is very easy to integrate in an EDA tool, needing only a module which dumps the internal automata data structures considering the syntax of the target language. Regarding to the interoperability between EDA vendors, our compiler supports the same Sugar subset as they do.

This work can be seen as a way of adding the power of formal verification into the classic verification based on simulators.

## Chapter 5 : Future work

The future work may cover the following aspects:

- The Sugar supported subset may be extended. Many problems may appear because the complement and deterministic automata for temporal expression automata will be needed.
- There is still work to be done to reduce the resulted automata (e.g. automata with fewer states, fewer transitions, and simpler predicates), in order to obtain the smallest and simplest possible. The fact that the automata size affects the resources required by the verification tools motivates this work. Figure X presents in opposition two automata for `cover {a[+]}` verification directive. First, the one generated by the compiler, is more complicated than the second, the one designed by hand.

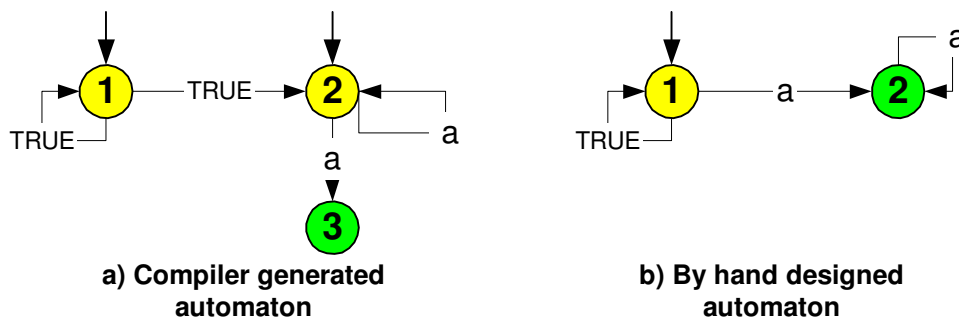


Figure 4 Automata for `cover {a[+]}` verification directive

- Different *Writer Automata Modules* may be developed in order to integrate the compiler in different verification tools.

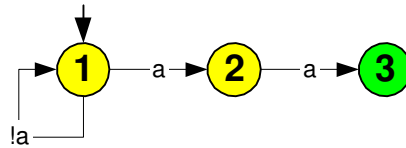
# Appendix A: Sugar Translation examples

Automata legend:



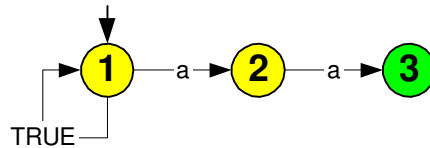
1. Tests if the first time when  $a$  is asserted is a pulse.

```
assert {!a[*]; a} ==> {!a};
```



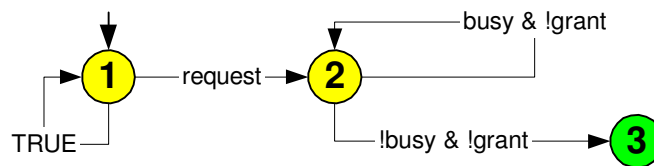
2. Tests if  $a$  is a pulse.

```
assert always {a} ==> {!a};
```



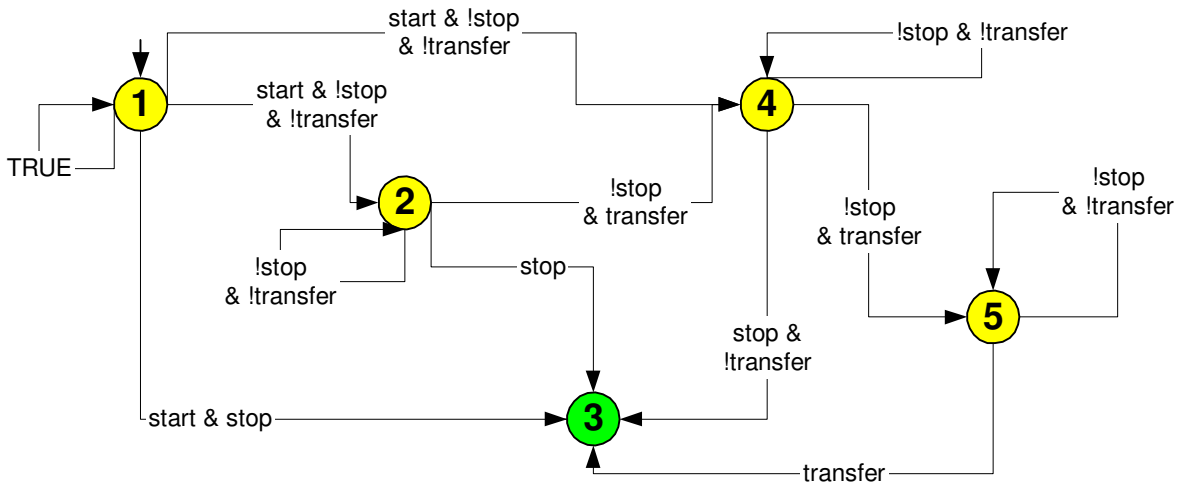
3. Tests if the after  $request$  is asserted,  $busy$  signal is asserted until  $grant$  is asserted.

```
assert always {request} ==> {busy[+]; grant};
```



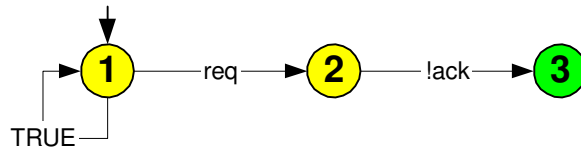
4. Tests if between a *start* signal assertion and the following *stop* signal assertion, the *transfer* signal is asserted for two cycles.

```
assert always {start} |-> {!stop[*];stop}&&{transfer[=2]};
```



5. Tests if after a *req* (request) an *ack* (acknowledge) follows on the next cycle.

```
assert always req -> next ack;
```



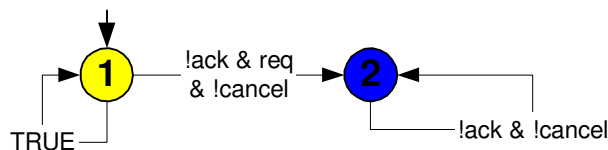
6. Tests if after a *req* (request) is always followed by an *ack* (acknowledge).

```
assert always req -> eventually! {ack};
```



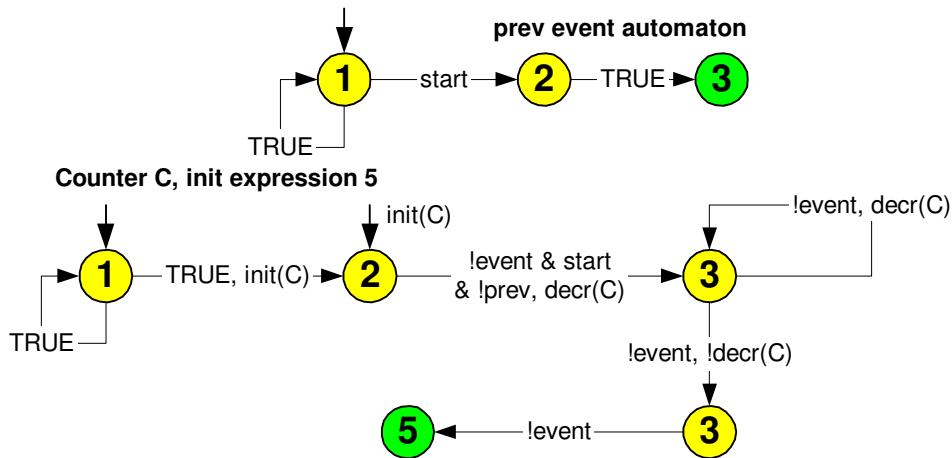
7. Tests if after a *req* (request) is always followed by an *ack* (acknowledge), unless a cancellation occurs.

```
assert always ((req -> eventually! {ack}) abort cancel);
```



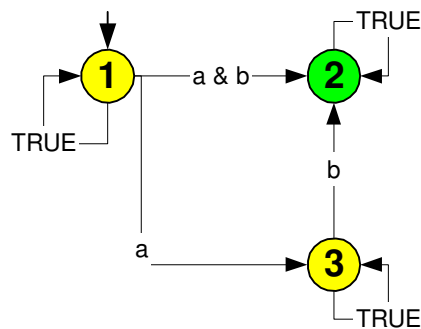
8. Tests if in the first five cycles after the *start* is asserted, the *event* is asserted.

```
assert always(rose(start)) -> next_e[0..5](event)
```



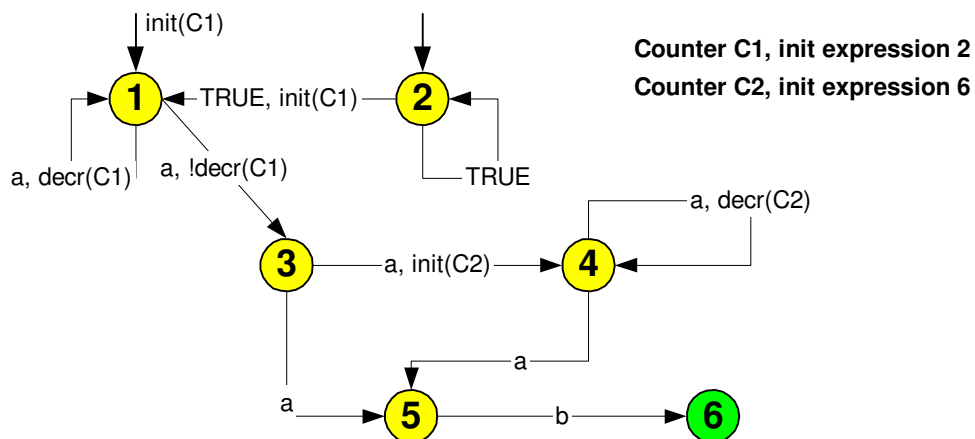
9. Signals always after events *a* and *b* were asserted..

```
cover {[*];b} & {a;[*]};
```



10. Signals always after events *a* and *b* were asserted.

```
cover {a[*3..9];b};
```



## Appendix B: Sugar Supported Grammar Subset

---

### Meta-Syntax:

```
Definition of a non-terminal: ::=
Alternative definitions:      |
Literals in single quotes:  '{'
Keywords in single quotes:   'automaton'
Non-terminals capitalized:   Automaton_Body
Semantic qualifiers in <>:   <module>
Optional parts in []:       [ Clock_Declaration ]
Zero or more repeats in {}: { Var_Declaration }
```

---

### # Verification Units

```
Sugar_Specification ::= { Verification_Unit }
Verification_Unit   ::= 'vunit' Name [ '(' <module>Name ')' ]
                    '[' { VUnit_Item } ']'
Name                 ::= Identifier
VUnit_Item           ::= | Sugar_Declaration | Verification_Directive
```

---

### # Sugar Declarations

```
Sugar_Declaration ::= Property_Declaration | Sere_Declaration
                  | Endpoint_Declaration | Clock_Declaration
Property_Declaration ::= 'property' Name '=' Temporal_Expression ';'
Sere_Declaration     ::= 'sequence' Name '=' '{' SERE '}' ';'
Endpoint_Declaration ::= 'endpoint' Name '=' '{' SERE '}' ';'
Clock_Declaration    ::= 'default' 'clock' ':' Boolean ';' 
```

---

### # Sugar Statements

```
Verification_Directive ::= Assert_Statement | Assume_Statement
                       | Cover_Statement
Assert_Statement       ::= 'assert' Temporal_Expression ';'
Cover_Statement        ::= 'cover' '{' SERE '}' ';' 
```

---

### # Sugar Temporal Expressions

```
Temporal_Expression ::= FL_Formula
FL_Formula ::= Boolean
            | '(' FL_Formula ')'
            | <property>Name
            | Boolean '->' FL_Formula
            | Boolean '<->' Boolean
            | FL_Formula 'abort' Boolean
            | FL_Formula '@' <clock>Boolean
            | '{' SERE '}' '(' FL_Formula ')'
            | 'always' FL_Formula
            | 'never' SERE
            | 'next_a' '[' Number '..' Number ']' FL_Formula
            | 'next_e' '[' Number '..' Number ']' Boolean
            | 'next' '[' Number ']' FL_Formula
            | 'next' FL_Formula
            | 'eventually!' SERE
```

```

| Boolean 'before' Boolean
| FL_Formula 'until' Boolean
| Boolean 'until_' Boolean
| '{' SERE '}' '|' -> '{' SERE '}'
| '{' SERE '}' '|' => '{' SERE '}'

```

---

### # Sugar Extended Regular Expressions

```

SERE ::= Boolean
      | '{' SERE '}'
      | <sere>Name
      | SERE ';' SERE
      | SERE ':' SERE
      | '{' SERE '}' AndOrOp '{' SERE '}'
      | SERE '[' '=' Range ']'
      | SERE '[' '->' Range ']'
      | SERE '[' '*' Range ']'
      | '[' '*' Range ']'
      | SERE '[' '+' ']'
      | '[' '+' ']'

```

```

AndOp ::= '&&' | '&' | '|'

```

```

Range ::= Number | Number '..' | '..' Number | Number '..' Number

```

---

### # Forms of Expression

```

Boolean      ::= <boolean>HDL_or_Sugar_Expression
Number       ::= <integer>HDL_or_Sugar_Expression
HDL_or_Sugar_Expression ::= HDL_Expression
                    | <endpoint>Name | Built_In_Function_Call
Built_In_Function_Call ::= 'rose' '(' Boolean ')'
                    | 'fell' '(' Boolean ')' | 'prev' '(' Boolean ')'

```

---

### Restrictions:

- Only single clock properties are accepted.
- Preferably do not use any repetition operators with counters in the right-hand operand of  $| \rightarrow$  and  $| \Rightarrow$ , because for that operand a deterministic automaton is generated.

## Appendix C: Sugar Preprocessor Macros

---

### **%for construct**

#### **Syntax:**

```
'%for' '(' identifier '=' expression ';' expression ';'
        identifier '=' expression ')'
...
'%end'

'%for' identifier 'in' expression '..' expression 'do'
...
'%end'

'%for' identifier 'in' '(' item ',' item ',' ... ',' item ')' 'do'
...
'%end'
```

where *item* is an integer, a real number or a quoted string.

**Description:** Replicates the piece of text a number of times, with the possibility of each replication receiving a parameter.

---

### **%switch construct**

#### **Syntax:**

```
'%switch' expression
'%case' expression ':'
...
'%case' expression ':'
...
'%default' ':'
...
'%end'
```

#### **Observations:**

- `%breaksw` may be used to break the active region.
- The usage of `‘:’` after `%case` and `%default` *is* optional.

**Description:** The `%switch` extension, and its matching `%end` are used to conditionally include code. The `%case` identifies the region which is evaluated. The active region is the one contained between the selected `%case` and the following one. The `%breaksw` macro may be used to break the active region.

---

---

## **%if construct**

### **Syntax:**

```
'%if' expression '%then'  
    ...  
'%else'  
    ...  
'%end'
```

**Description:** The %if expression allows expressions to be evaluated, and if true, will active the region of code between the %if and the matching %end (or the matching %else, if it exists).

---

## **%while construct**

### **Syntax:**

```
'%while' expression  
    ...  
'%end'
```

**Description:** The code between the %while and its matching %end is the region that is repeated until the expression is not true.

---

## **%let construct**

### **Syntax:**

```
'%let' identifier '=' expression
```

**Description:** The %let is used to assign variables. Variables can be integers, real numbers, or strings. The type of the variable is determined by the resulting operations. Any identifier that is assigned with a %let is available for evaluation in any subsequent expressions.

**Observations:** Currently, operators on strings are not defined (e.g. Two strings can not be added).

---

## **Expressions**

**Description:** Expressions can be simple or complex and include the following operators: numeric, logical bitwise, equality and functions. Precedence rules follow those of Verilog (e.g. multiplication has higher precedence than addition). The use of brackets ("(" and ")") allows user specification on order of evaluation.

**Numeric operators:** addition (+), subtraction (-), multiplication (\*), division (/), modulus (%), power (\*\*).

**Logical operators:** logical AND (&&), logical OR (||), logical NOT (!).

**Bit operators:** bitwise XOR (^), bitwise AND (&), bitwise OR (|), shift right (>>), shift left (<<).

**Equality operators:** equality (==), inequality (!=), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=).

**Function:** log2 (`LOG2()`), round up (`CEIL()`), round down (`FLOOR()`), round to nearest value (`ROUND()`), max of two numbers (`MAX()`), min of two numbers (`MIN()`), odd (`ODD()`), even (`EVEN()`), absolute value (`ABS()`).

---

**Variable value access:**

Any created variable can be expanded as a string in the Sugar code using the following format: `%{identifier}`.

The value of an expression may be evaluated and copied in the expanded Sugar code using the following format: `%{expression}`.

---

## References

1. Sugar 2.0, Proposal Presented to the Accellera Formal Verification Technical Committee, Cindy Eisner, Dana Fishman,  
[http://www.haifa.il.ibm.com/projects/verification/sugar/Sugar\\_2.0\\_Accellera.ps](http://www.haifa.il.ibm.com/projects/verification/sugar/Sugar_2.0_Accellera.ps)
2. Sugar 2.0 Early Interoperability Guidelines, Erich Marschner
3. A. Pnueli. A temporal logic of concurrent programs, Theoretical Computer Science, 13:45-60, 1981
4. E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic, Proc. Workshop on Logics of Programs, LNCS 131, pages 52-71, Springer-Verlag, 1981
5. IEEE Standard 1364-2001, 2001.
6. Cpp, GNU C Preprocessor, <http://gcc.gnu.org/projects/cpplib.html>.
7. Himanshu M. Thaker, Vpp, A Verilog Preprocessor,  
<http://www.surefirev.com/vpp/vbpps-1.0.html>.
8. Ilan Beer, Shoham Ben-David, Cindy Eisner , Dana Fisman, Anna Gringauze, Yoav Rodeh, The Syntax and Semantics of Sugar.
9. Ilan Beer, Shoham Ben-David, Cindy Eisner , Dana Fisman, Anna Gringauze, Yoav Rodeh, The Temporal Logic Sugar.
10. George Vastianos, Boolean functions' minimisation software based on the Quine-McCluskey method, Software Notes, Athens-GR, April 1998.
11. David Belton, Minimisation of Boolean Functions, Combinational Logic & Systems Tutorial Guide, University of Surrey, Surrey-UK, April 1998.

