A C-language binding for PSL

Ping Hang Cheung, Alessandro Forin
Microsoft Research

September 2006

Technical Report
MSR-TR-2006-131

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

A C-language binding for PSL

Ping Hang Cheung, Alessandro Forin

Microsoft Research, One Microsoft Way, Redmond, WA, USA
cheung@cecs.pdx.edu, sandrof@microsoft.com

Abstract. In recent years we have seen an increase in the complexity of embedded system design and in the difficulties of their verification. As a result, engineers have been trying to verify the specifications at a higher level of abstraction. In this paper we present an automated tool which is able to perform runtime verification of a program's logical properties asserted by the programmer. The idea is to leverage the Assertion Based Verification language PSL, which is widely used by hardware engineers, extending it to the software verification of C language programs. The properties expressed in a simple subset of PSL are evaluated by the tool during full-system simulation. Like in hardware Assertion Based Verification, the tool can handle both safety properties (absence of bad events) and liveness properties (good events eventually happen). The liveness property is not widely supported in existing verification tools.

Keywords: Property Specification Language, C, Assertion Based Verification

1. Introduction

Assertions Based Verification (ABV) is an approach that is used by hardware design engineers to specify the functional properties of logic designs. Two popular languages based on ABV are the Property Specification Language PSL and the System-Verilog Assertion system SVA [1]. PSL is now an IEEE standard – P1850 [2]. PSL specifications can be used both for the design and for the verification processes. A single language can be used first for the functional specification of the design and later on as an input to the tools that verify the implementation. The backbone of PSL is Temporal Logic [3], [4]. Temporal Logic can describe the execution of systems in terms of logic formulas augmented by time-sequencing operators.

In this paper, we introduce a binding of PSL to the C programming language. A programmer can write PSL statements about her C program and the properties are verified during execution. Our initial work has shown that the approach is feasible; we have defined a simple subset of PSL (sPSL) and realized a few tools with which we can perform ABV of C programs using Linear Temporal Logic (LTL). The sPSL LTL operators are provided for describing events along a single computation path.

sPSL is implemented using the Giano simulator [5] as the execution platform. Giano is a dual-headed hardware-software simulator. It is capable of performing the full-system simulations of CPUs and hardware peripherals as well as the behavioral simulation of hardware designs written in Verilog. The sPSL engine is realized modifying an existing ARM CPU simulation module from Giano.

The rest of the paper is structured as follows. Related work in the verification field is discussed in Section 2. Section 3 introduces the sPSL language. The architecture of the sPSL execution engine is described in Section 4 and Section 5 provides some simple examples. Section 6 concludes with a discussion of improvements we have planned for further assessments of the sPSL capabilities.

2. Related Work

LTL properties can be translated into code that is added to the target program to monitor it during execution, as with the Temporal Rover and DBRover tools [6, 7]. Temporal Rover is a code generator which accepts source code from Java, C, C++, Verilog or VHDL. The LTL assertions are expressed as comments embedded in the source code. With the aid of a parser, the assertions are inserted in the source code that is then compiled and executed.

Java-MaC [8] is a more limited system, restricted only to Java programs. It contains a static phase and a run-time phase. At program analysis time, it uses the Primitive Event Definition Language (PEDL) to define events and their desired relationships. At run-time, it continuously monitors and checks the executing program with respect to the defined formal specifications. An even simpler approach to detect software faults at runtime is to use a pre-processor and assertions, as with ASAP [9]. ASAP is a pre-processor for C programs, it extends the usage of assertions in C programs by using partial functions and first order logic. Inevitability, these assertions are embedded in the program source code.

Rosu [10] suggests re-writing techniques to evaluate LTL formulas. The execution of an instrumented program creates traces of interesting events and the rewriter operates on such traces. Some algorithms assume the entire trace is available for (backward) analysis, others can process each event as it arrives. Rosu's algorithms make it possible to generate very efficient monitors that can be used by practical tools such as the Java PathExplorer (JPaX) [11].

In Design by Contract, a class specification is augmented with behavioral specifications. The user (client) must agree both to the syntactic requirements and to the behavioral requirements in order to invoke a method specified by such a class. One instance is the Java Modeling Language (JML) [12]. JML is a behavioral interface specification language for Java modules. The JML Compiler (jmlc) compiles JML code into runtime checks of the class contracts. In [13], the jmlc compiler is used in conjunction with an Extended Static Checker for Java version2 (ESC/Java2). In [14] this approach is used to perform verification of a full compiler. ESC/Java2 makes additional use of static analysis, a technique that does not require actually executing the program for fault detection. Another instance is Spec# [15]. The Spec# programming language is a superset of C# which provides method contracts in the form of pre-conditions and post-conditions, as well as object invariants. The Spec# compiler provides run-time checking for method contracts and object invariants. A Spec# static program verifier generates the logical verification for Spec# program and an automated theorem prover analyzes the verification directives to prove the program's correctness.

SLIC [16] is a language for specifying the low level temporal safety properties of Application Program Interfaces (APIs) defined in the C programming language. It can be used along with the companion tool SLAM [17] to perform validation. Our approach is similar, we also use a specification language and a verification tool as the two key components for validation. In our case, sPSL is the language for specifying the program properties and the Giano simulator augmented with the sPSL evaluation engine is the verification tool.

All of these systems insert instrumentation code into the executing program to monitor and check events and therefore introduce some execution overhead that can potentially modify the program's temporal behavior. This is not acceptable for Real-Time programs and even a limited overhead is poorly received by developers. In our approach the program binary is not modified in any way, the monitoring is performed entirely by the execution engine (the Giano simulator).

3. sPSL

There are two layers supported by the current implementation of sPSL: the Temporal Layer and the Verification Layer. The complete PSL specification includes also a Modeling layer which we did not implement in sPSL. The Modeling layer is typically used for modeling external inputs.

3.1 Temporal Layer

A program can be described by temporal expressions. A temporal expression involves events that are ordered by timing relationships. With the aid of temporal expressions, we can define properties that describe the behavior of a program in a machine readable form.

3.2 Verification Layer

sPSL is described in an external file rather than being embedded in the C source file. A *verification unit*, or *vunit*, in PSL links the scope of function and variable names back to the C program. A vunit is also the syntactic container for the sPSL properties. A vunit is checked at runtime when the corresponding basic blocks in the C program are activated. A *vunit* takes an argument that identifies the lexical scope where local variable names are bound. The argument is in the form shown in equation (1)

vuint_argument == filename [:: function_name [block]]; (**1**)
block == { [block] } [block];

Specifying only the filename makes visible all functions and global variables visible during the compilation of that specific source file (global scope). Specifying the function name adds to the global scope the parameters of that function, but none of the local variables. Notice that according to the C rules a parameter will overrule a global variable of the same name. Specifying a left-bracket adds to the scope all variables at the outmost lexical scoping level within that function. Specifying more than one left-bracket identifies blocks that are further indented. Numbers can be used to shorten a block identifier.

Properties in the temporal layer are expressed as declarations in the PSL language. In order to validate the system we need to use *verification directives* that specify how/when those properties hold true. Since we are considering simulation based verification, formal verification flavored units like *assume* are not currently covered. *Assume* statements specify the values of input variables for use by a formal verification tool. *Assert* is the only verification unit currently supported by sPSL. It tells the verification engine to check whether the assertion of a property holds. If a property fails to meet the requirements, an error will be reported to the user.

In the sPSL shown below, the fragments *vunit check_foo (foo.c::baz)* and *assert always_foo* represent the verification layer while the *property always_foo = always (foo=1)* represents the temporal layer. The verification directive *assert* guarantees that *always_foo* holds valid for the life of the block, which in this case includes the whole program. The syntax for the declarations and the functionality of each operator are described in the following.

```
vunit check_foo(foo.c)
{
      property always_foo = always (foo=1);
      assert always_foo;
}
```

3.3 Declarations

Each sPSL property declaration is introduced by an identifier for that property, such as *always_foo*. The property is then followed by an expression involving one or more operators. Time advances monotonically along a single path, left to right through the expression. Only a subset of the PSL operators is included in sPSL, taken from the Foundational Language (FL) subset. Valid sPSL operators are *always*, *never*, *eventually*, *until*, *before* and *next*. Each operator belongs to an operator class. For instance, *always* and *never* are the FL invariance operators, *eventually* and *until* are the FL occurrence operators, *before* and *next* are the bounding operators. In order to express the liveness properties we also support the operators *eventually*!, *until*!, *before!* and *next*!.

3.4 Operators

The operator *always* guarantees that a temporal expression will hold true. In the property shown in (2), the variable *const* is required to always hold the value "1". Assuming that *const* is a global variable *always* in this case means for the entire life of the program. If the expression instead refers to local variables then the property will be checked only while those variables are in scope, meaning for the duration of the function call.

```
property  check_always = always (const = 1);
```
(2)

The operator *never* guarantees that the expression will never become true. In (3), the specified assignment of "0" to the variable *const* must not happen during the life of the program.

```
property check_never = never (const = 0);
```
(3)

The operator *until* guarantees that an expression is true until a second expression becomes true.

```
(arg1 until arg2)
```
(4)

```
(arg1 until! Arg2)
```
(5)

Let us illustrate the execution diagram in Figure 1 and the properties shown in (4). In the execution depicted in Figure 1, the property is validated because *arg1* is true until *arg2* becomes true at time t=x. There are two variations of this operator, *until!* and *until*. The operator *until!* shown in (5) is a strong operator used to indicate that *arg2 must* eventually become true in order to satisfy the property and it is an error if this never happens. The operator *until* is a weak operator used to indicate that *arg2* can satisfy the property. If *arg2* is never true this is not an error, provided that *arg1* is true.

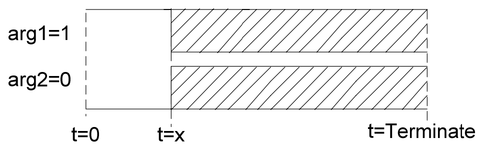


Figure 1 Timing for until

The operator *before* guarantees that an expression is true before a second expression becomes true.

```
(arg1 before arg2)
```
(6)

```
(arg1 before! arg2)
```
(7)

The execution diagram in Figure 2 shows that the property of `(6)` is validated because *arg1* is true before *arg2* becomes true. In contrast to the previous examples for the *until* operator, *arg1* here is not required to hold at all times, but only at least once before *arg2* becomes true at time t=y.

Again there are two variations of this operator - *before* and *before!*. The strong operator *before!* shown in (7) requires that the expression *arg1* (rather than *arg2*) will eventually become true and it is an error if this never happens. This is not an error instead for the operator *before*. For both operators, it is a violation if *arg1* is never asserted before *arg2*.

```
arg1 before arg2 = not(arg2) until not(arg1)
```
(8)

Note that while the operator *before* and *until* are equivalent according to the relation of (8), this does not hold for the operators *before!* and *until!*.

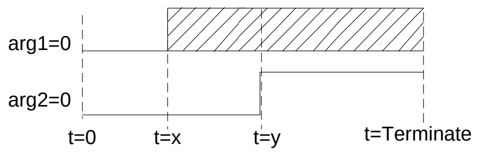


Figure 2 Timing for before

The operator *next* guarantees that an expression will hold true in the next execution cycle. The existing prototype supports the use of the *next* and *next!* variants.

```
always (arg1 ->next arg2)
```
(9)

This operator is slightly different from the original PSL definition, which referred to a concept of system clocks and cycle counts that is not directly applicable to software. In sPSL the "next execution cycle" means rather "the next event". We use *next* to require that if *arg1* becomes true then in next assignment that affects any of the logic properties it will be *arg2* that becomes true, or in other words that *the next interesting event* is that *arg2* becomes true at t=x. The operator *next!* is used in the same way as

PSL, to require that if *arg1* becomes true then *arg2* will eventually become true as well. This operator can be useful when dealing with critical sections of code where the processor cannot be interrupted.
Figure 3 and (9) illustrate the use of next, with the assumption arg1=0 and arg2=0 initially.

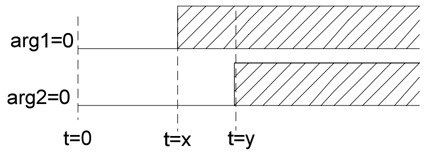


Figure 3 Timing for next

The operator *eventually!* guarantees that "something good" eventually happens. In (10), if the expression *arg1* becomes true then there must be an execution path which leads to *arg2* also becoming true sometimes in the future. Consider the diagram described in Figure 4, where time advances to the right. The shaded area in the figure represents "don't care" values, the variable could be either 0 or 1. Assume the program starts the execution at t=0 and terminates at t=Terminate with the initial condition arg1=0 and arg2=0. At time t=x "*arg1*" becomes true. If at time t=y, *arg2* becomes true we can claim that the "*check_evenutally*" property is valid, even if *arg1* should become false between t=x and t=y. It is indeed a violation of the property if *arg2* never becomes true after time t=x and before t=Terminate.

```
always (arg1 ->eventually! arg2)
```
(10)

Note that always is also part of this property specification; we require checking for the entire life of the program.

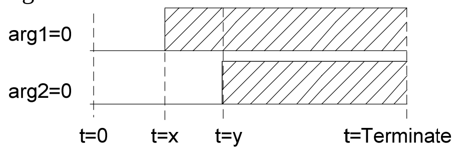


Figure 4 Timing for eventually

4. Evaluation

There are two separate components that make up our implementation, namely the data model generator and the evaluation engine. The data model generator is responsible for processing data from the sPSL source, C source file, and from a textual dump of the debugging information contained-in/related-to the executable image and collect it into a single file for later use by the evaluation engine.

The evaluation engine has two interfaces, one to the execution processor and one to the data model. It retrieves from the execution processor such information as memory addresses, instructions, and register contents. It uses the data generated by the data model generator to realize the desired property checking. Figure 5 depicts the architecture of the prototype.

4.1 Data Model Generation

The sPSL source is processed first by a script to create one entry for each property declared in the sPSL source file. After processing the C source file the model will also contain a tag for each of the variables and functions found in the C source. The C source is compiled and the compiler is instructed to generate maximum debugging information. This information is extracted into a text file by compiler tools such as OBJDUMP or similar. The data model generator reads that information and adds to the data model the addressees and offsets of functions and variables, register allocation information and the values of some individual instructions. The data model also contains the start and end addresses of the basic blocks, which are needed to recognize the entering and exiting of the scope of local variables. If the image is actually executed at a different load address (runtime relocation) an offset is added to the statically identified information. Some additional information is needed for sPSL operators with two operands, for these operators the data model will identify their *insertion points* and *release points*. An *insertion point* is the set of execution addresses that affect the left-side operand of the operator, for instance the point at which all variables are in scope and the property is live or when a specific variable is modified. Conversely, the *release point* is the set of execution addresses that affect the right-hand operand, for instance when a function returns and some variables are no longer in scope.

The data model itself is a human-readable text file, the evaluation engine will later parse it to create a more efficient representation in the form of individual decision tree (PTree).

4.2 Evaluation engine

The sPSL evaluation engine is a module that is physically part of the Giano simulator and monitors the instruction addresses, memory references, and registers accesses during program execution. Every time a new program is launched during execution, the runtime system notifies the Giano simulator of the program name and the address at which it was loaded. The evaluation engine uses the program name to look for a corresponding data model file, if it finds it it parses it and creates the corresponding PTree. When a specific property is live, the engine creates and initializes an evaluation tree for that individual property (ETree) and the monitoring task is started.

For instance, assume that some property is defined by the operator *never* and that the expression refers to a single global variable. The evaluation engine will monitor all memory references looking for stores to that specific memory location. If the new value assigned to the variable violates any property an error is immediately reported in the Giano execution window. In general, weak operator report errors immediately and strong operators report errors when the scope exits. Program termination is a case of exiting scope and it also affects all global variables.

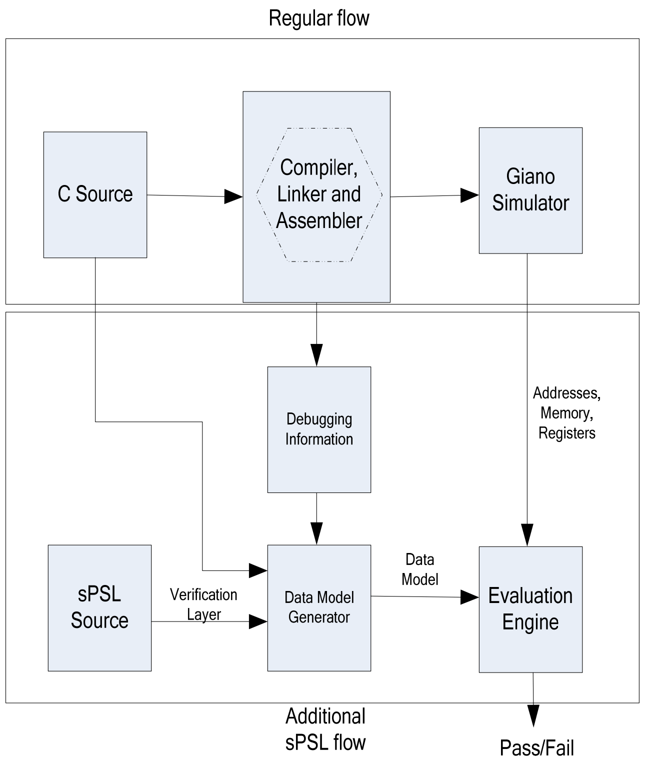


Figure 5 Architecture of the Prototype

4.3 Tree Evaluation

The evaluation of the ETree is performed with a depth-first, left-first traversal.. Each branch/sub-branch corresponds up to 2 leaves. These leaves contain either a value or an operator. We use ternary logic during the evaluation, with the values true (T), false (F) and undefined (Z). An example of a property and the corresponding ETree is shown in Figure 6.

In Figure 6, the node *a=1* is the first insertion point for the property. Assume that this expression becomes true at time t. The parent node is a *next* operator, we need to wait

until the next event to be able to decide whether the operator is satisfied or not, therefore we return Z. If the next event is indeed an assignment of “1” to the variable *b* the *next* operator can return T. If instead the variable is “0” an F is returned. Either way the operator *next* can now return a defined value.
Once the parent node *until* receives a “*T*” from the left-side subtree it can monitor the release point for the right-hand subtree, namely *c=1*. Until the *c=1* is satisfied we return Z. Once *c=1* and provided that *a=1 next b=1* still hold, the *until* can return T to the parent node *always*.

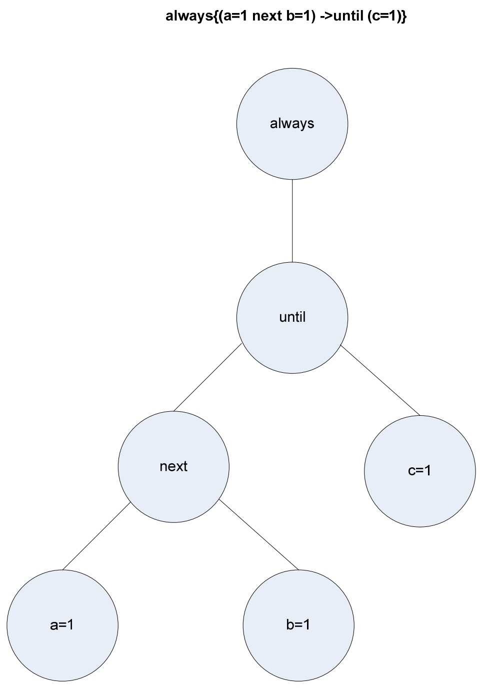


Figure 6 A Property and its Evaluation Tree

The invariance operator *always* cannot return a definite value until termination, which is either the exiting of the scope or program termination. The event of its operand becoming true does have an effect though, logically the property is satisfied and immediately re-instated. Evaluation restarts then from the initial state.
Notice that when a subtree reports an F this is not cause for failure, only if this happens at the top of the tree. A simple counter-example is “not (a=1)”.

In the following, we will describe how the various operators are implemented.

	Variable		Function invocation	
	Left-op	Right-op	Left-op	Right-op
eventually!	Yes	Yes	Yes	Yes
until/until!	Yes	Yes	No	Yes
before/before!	Yes	Yes	No	Yes
next/next!	Yes	Yes	Yes	Yes

Table 1

Not all combinations of function invocations and variable references are allowed in expressions, as indicated in Table 1. Specifically, all types of *until* and *before* operators do not allow a function invocation as a first argument. Unlike variables, a function invocation doesn't hold onto a value for any set period of time

EVENTUALLY!
Every function invocation and every store to variables that appear in *arg1* are insertion points for the *eventually!* operator. Variables and functions in *arg2* are release points. When a function invocation occurs, the stack frame determines the addresses of local variables and can make a property live. This activates the insertion points and release points. The engine is invoked at insertion points and checks to see if the new value makes the expression true. Once that happens, further insertion points are ignored and only release points are monitored. Once a release point is reached that renders true the expression *arg2* the property is satisfied and no further monitoring is required. If the release point is never reached before the end of the scope an error is generated.

UNTIL/UNTIL!
The operator *until* is similar to eventually as far as insertion and release points are concerned. However, once *arg1* is satisfied for the first time the insertion points are not released, they are still used to verify that *arg1* holds until the release point is reached or the scope is exited. An assignment that renders *arg1* false before the release point is a violation of the property. Once a release point is reached that renders true the expression *arg2* the property is satisfied and no further monitoring is required. If the release point is not reached when exiting the scope the property is violated, but only for the strong operator *until!*.

BEFORE/BEFORE!
The operator *before* is similar to the operator *until*, except the roles of the expressions *arg1* and *arg2* are swapped and negated. Notice however that the *before!* operator requires that *arg2* eventually holds, which is not the case for *arg1* in *until!*.

NEXT/NEXT!
The operator *next* is very similar to the operator *eventually*!. The insertion points are the same. The release points however are only evaluated in a specific moment in time and then released, not constantly as is the case for *eventually!*.

5. Examples

```
1: int main()
2:{
3:        UINT32 addr1 = 1;
4:        UINT32 addr2 = 2;
5:        UINT32 INTR = 0;
6:        UINT32 op = 0;
7:
8:        send_to_HW(addr2,0x0,0x3);
9:
10:       while(1)
11:       {
12:         INTR=TheBCTRL->GCTRL_out;
13:         if(INTR == 1)
14:         {
15:           op=5;
16:           send_to_HW(addr1, addr2, op);
17:           break;
18:         }
19:       }
20:       return(0);
21:}
```

The partial code shown above is a Real-Time C program with two simple steps. On line 8 the function call to *send_to_HW(addr2,0x0,0x3)* affects a certain peripheral hardware, which is expected to trigger an interrupt in return. On line 13, if *INTR* is 1 it means that the interrupt has indeed happened.

```
vunit check_intr(example.c::main)
{
```

```
property intr_event = always (send_to_HW(addr2,0x0,0x3)
->eventually! INTR=1)
assert intr_event;
}
```

In the above code, we create a property *intr_event* to monitor that *INTR* eventually happens. The left operand *send_to_HW(addr2,0x0,0x3)* is marked as insertion point and *INTR=1* is marked as release point. When the insertion point is satisfied, the evaluation engine will monitor the release point. Before the release point holds, the *eventually!* node returns a "*Z*". It returns a T only once the right operand holds. Iff the right operand does not hold until the scope exits the property fails.

```
1:  int i=0;
2: char buffer[10];
3:
4: int main()
5:{
6:        while(1)
7:        {
8:          i++;
9:          buffer[i]=1;
10:       }
11:       return 0;
12:}
```

The partial code shown above is a general purpose C program. On line 9, a buffer overflow error will occur if the index into the buffer exceeds 10.

```
vunit check_overflow(example.c)
{
property overflow = never((i>10) OR (i < 0));
assert overflow;
}
```

The above sPSL code shown the property "overflow" monitors the increment of *i*. The operator *never* holds the value "*T*" if *0<i<10*. Otherwise, "*F*" is returned.

6. Conclusion and Future Work

The first prototype of sPSL shows that it is possible to use a simple subset of the Property Specification Language PSL to perform assertion based verification of C language programs. To our knowledge, this is the first time that PSL, an IEEE-standard language widely used for hardware verification, has been applied to software programs.

The approach we used, namely to use a modified full-system simulator to execute the program, has not been used before for the verification of software programs. The main advantage of this approach is that no modifications are made to the executable program and no additional instrumentation code is required, thereby increasing the confidence in the verification process itself.

The prototype generates execution traces in terms of function calls and variable changes that are useful to the programmer to understand the reason for the erroneous behavior. The traces could also be used by other tools for further analysis, such as performance analysis and assessment of execution time boundaries.
The tool already supports real-time specification and this could be used for performance verification as well.
The sPSL language and the evaluation engine do not depend on the particular programming language we used, they would apply just as well to any block-structured language implemented by a stack-register architecture. It should therefore be possible to extend sPSL to other languages like C# or even FORTRAN simply by creating the corresponding programming language parser. Similarly for a different processor like the PowerPC or the MIPS.

The current prototype does not provide support for the Sequential Extended Regular Expressions (SERE). Within the FL operators, suffix implication and partial logical implication will be implemented for the next prototype.
The current prototype supports only the equality operator in Boolean expressions, and furthermore expressions can only refer to a single variable.

We have made no attempt at this point to quantify and/or minimize the overhead in execution time due to the sPSL engine.
One possible extension of this work is to attack the problem of mixed software-hardware verification. Giano would appear to be a promising tool in this regard. Fei Xie's xPSL [18] is one project that is trying to find a unified solution to the problem.

References

[1] Accellera and I. 1364, "SystemVerilog."
[2] Accellera, "IEEE P1850 PSL."
[3] A. N. Prior, *Past, Present and Future*: Oxford University Press, 1967.
[4] A. Pnueli, "The temporal logic of programs," *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77),* pp. 46-57, 1977.
[5] A. Forin, B. Neekzad, and N. L. Lynch, "Giano: The Two-Headed System Simulator," *Microsoft Research Technical Report,* vol. MSR-TR-2006-130, 2006.
[6] D. Drusinsky, "The Temporal Rover and the ATG Rover," *Proc. of SPIN'00: SPIN Model Checking and Software Verification,* vol. 1885, pp. 323-330, 2000.

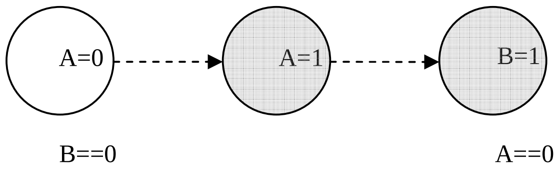
[7] D. Drusinsky, "Monitoring Temporal Rules Combined with Time Series.," *Proc. of CAV'03: Computer Aided Verification,* vol. 2725, pp. 114-118, 2003.
[8] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, "Runtime assurance based on formal specifications," *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications,* 1999.
[9] I. D. D. Curcio, "A Simple Assertion Pre-processor," *SIGPLAN,* vol. 33, pp. 44-51, 1998.
[10] G. Rosu and K. Havelund, "Rewriting-based Techniques for Runtime Verification," *J. of ASE,* vol. 12, pp. 151-197, 2005.
[11] K. Havelund and G. Rosu, "Java PathExplorer --- A runtime verification tool," *Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space, ISAIRAS'01, Montreal, Canada,* 2001.
[12] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Muller, J. Kiniry, and P. Chalin, "JML Reference Manual," 2006.
[13] P. Chalin and P. James, "Cross-Verification of JML Tools: An ESC/Java2 Case Study," *Microsoft Research Technical Report,* vol. MSR-TR-2006-117, 2006.
[14] P. Chalin, C. Hurlin, and J. Kiniry, "Integrating Static Checking and Interactive Verification: Supporting Multiple Theories and Provers in Verification," in *VSTTE 2005*, 2005.
[15] K. R. M. L. Mike Barnett, Wolfram Schulte, "The Spec# programming system: An overview," *CASSIS 2004, LNCS* vol. 3362, 2004.
[16] T. Ball and S. K. Rajamani, "SLIC: A Specification Language for Interface Checking (of C)," *Microsoft Research Technical Report,* vol. MSR-TR-2001-21, 2001.
[17] T. Ball and S. K. Rajamani, "The SLAM Project: Debugging System Software via Static Analysis," in *POPL 2002*, 2002.
[18] F. Xie, X. Song, H. Chung, and R. Nandi, "Translation-based co-verification," *3rd ACM IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005),* pp. 111-120, 2005.

Appendix

This Appendix describes the semantic of the sPSL operators. Each operator is formally described by a LTL Temporal Logic formula, a sPSL property, an event diagram and an action table. Event diagrams depict events inside nodes and arrows indicating the flow of time. A solid arrow represents an “immediately follows” relationship, whereby events must be separated exactly by the indicated units of time. A dashed arrow indicates that an unspecified, possibly zero number of time units separates the two events. A dark node indicates that the operator is satisfied. A clear node indicates that the operator is not satisfied. The action table describes the sPSL engine actions at Scope entry and exit, insertion and release points, and repeated occurrences, e.g. within the *always* operator.

Operator:	*until*
LTL Formula:	*aWb*
sPSL Formula:	*a until b*
sPSL type:	FL bounding operator

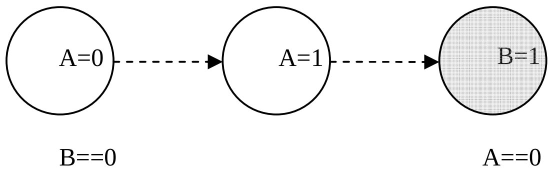
A until B



Condition	sPSL Engine Actions
Scope entry	Wait until all variables are initialized then create the insertion/release points. Evaluate both operands against these initial values.
Insertion point	Evaluate the left operand against the new value. Once it becomes true the operator is satisfied.
Release point	Evaluate the right operand against the new value. Once it becomes true the operator is satisfied iff the left operand is false. Otherwise the property is violated.
Repetition	Either the left or the right operand shall be true, but not both.
Scope exit	If the operator was never satisfied it is not a violation. Remove all insertion/release points.

Operator:	*until!*
LTL Formula:	*aUb*
sPSL Formula:	*a until! b*
sPSL type:	FL bounding operator

A until! B

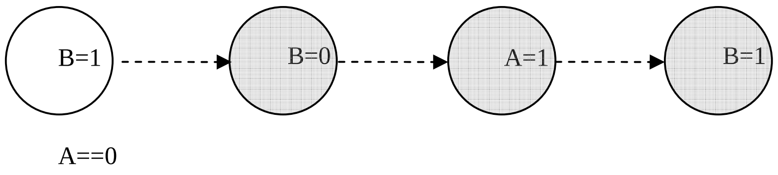


Condition	sPSL Engine Actions
Scope entry	Wait until all variables are initialized then create the insertion/release points. Evaluate both operands

	against these initial values.
Insertion point	Evaluate the left operand against the new value.
Release point	Evaluate the right operand against the new value. Once it becomes true the operator is satisfied iff the left operand is false. Otherwise the property is violated.
Repetition	Either the left or the right operand shall be true, but not both.
Scope exit	If the operator was never satisfied report violation. Remove all insertion/release points.

Operator:	*before*
LTL Formula:	*aRb*
sPSL Formula:	*a before b*
sPSL type:	FL bounding operator

A before B

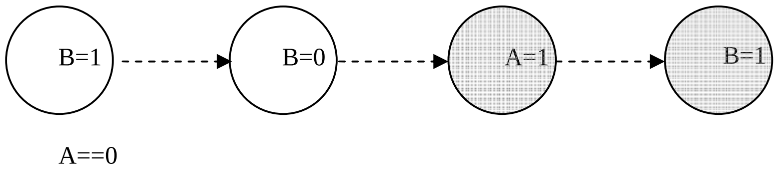


Condition	sPSL Engine Actions
Scope entry	Wait until all variables are initialized then create the insertion/release points. Evaluate both operands against these initial values.
Insertion point	Evaluate the left operand against the new value. Once it becomes true the operator is satisfied iff the right operand is false.
Release point	Evaluate the right operand against the new value. Once it becomes true the operator is satisfied iff the left operand is true. Otherwise the property is violated. Being false is not a violation.
Repetition	The right operand can only transition to true while the left operand is true.
Scope exit	If the operator was never satisfied it is not a violation. Remove all insertion/release points.

Operator:	*before!*
LTL Formula:	*aRb*

sPSL Formula:	*a before! b*
sPSL type:	FL bounding operator

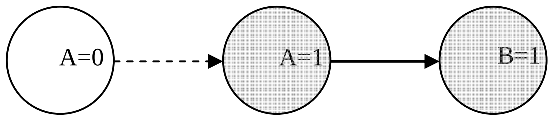
A before! B



Condition	sPSL Engine Actions
Scope entry	Wait until all variables are initialized then create the insertion/release points. Evaluate both operands against these initial values.
Insertion point	Evaluate the left operand against the new value. Once it becomes true the operator is satisfied iff the right operand is false.
Release point	Evaluate the right operand against the new value. Once it becomes true the operator is satisfied iff the left operand is true. Otherwise the property is violated.
Repetition	The right operand can only transition to true while the left operand is true.
Scope exit	If the operator was never satisfied report violation. Remove all insertion/release points.

Operator:	*next*
LTL Formula:	*aXb*
sPSL Formula:	*a next b*
sPSL type:	FL occurrence operator

A next B

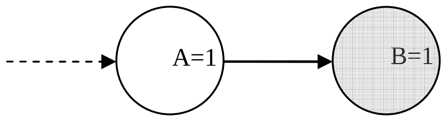


Condition	sPSL Engine Actions

Scope entry	Wait until all variables are initialized then create the insertion/release points. Evaluate the left operand against these initial values.
Insertion point	Evaluate the left operand against the new value. Once it becomes true the operator is satisfied. Mark the release points with a counter=1.
Release point	Evaluate the right operand against the new value. Once it becomes true the operator is satisfied iff the release points are marked with counter=1. Otherwise the property is violated. Unmark the release points.
Repetition	Every time the left operand is true the very next event must be the right operand becoming true, or scope exit.
Scope exit	If the operator was never satisfied it is not a violation. Remove all insertion/release points.

Operator:	*next!*
LTL Formula:	*a X! b*
sPSL Formula:	*a next! b*
sPSL type:	FL occurrence operator

A next! B

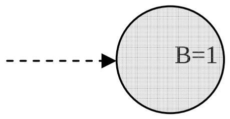


Condition	sPSL Engine Actions
Scope entry	Wait until all variables are initialized then create the insertion/release points. Evaluate the left operand against these initial values.
Insertion point	Evaluate the left operand against the new value. Once it becomes true mark the release points with a counter=1.
Release point	Evaluate the right operand against the new value. Once it becomes true the operator is satisfied iff the release point is marked with counter=1. Otherwise the property is violated. Unmark the release points.
Repetition	Every time the left operand is true the very next event must be the right operand becoming true.

Scope exit	If the release points are still marked with counter=1 report a violation. Remove all insertion/release points.

Operator:	eventually!
LTL Formula:	Fb
sPSL Formula:	eventually! b
sPSL type:	FL occurrence operator

eventually! B



Condition	sPSL Engine Actions
Scope entry	Wait until all variables are initialized then create the release points. Evaluate the right operand against these initial values.
Release point	Evaluate the right operand against the new value. Once it becomes true the operator is satisfied, remove all release points.
Repetition	For every instance where the right operand does not hold, the release point is marked as fail. If the right operand holds at least once, it will ignore the following right operand regardless it holds or not.
Scope exit	If there are still release points report a violation.

Operator:	->
Boolean Formula:	*if a then b*
sPSL Formula:	*a -> b*
sPSL type:	Logical IF implication

The logical implications are:

a	**b**	**a->b**
F	F	T
F	T	T
T	F	F
T	T	T

Condition	sPSL Engine Actions

Scope entry	Wait until all variables are initialized then create the insertion/release points. Evaluate both operands against these initial values.
Insertion point	Evaluate the left operand against the new value. Once it becomes true, proceed to release point.
Release point	Evaluate the right operand against the new value. Once it becomes true the operator is satisfied. Notice that this operator has to nest with the occurrence operator. The left operand is the starting condition for the nested occurrence operator.
Repetition	The right operand (occurrence operator) can only transition to true while the left operand is true.
Scope exit	Depends on the operator in the right operand.