The Design and Implementation of P2V, An Architecture for Zero-Overhead Online Verification of Software Programs

Hong Lu
Texas A&M University

Alessandro Forin
Microsoft Research

August 2007

Technical Report
MSR-TR-2007-99

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

The Design and Implementation of P2V, An Architecture for Zero-Overhead Online Verification of Software Programs

Hong Lu
Department of Computer Science
Texas A&M University
College Station, TX 77843
+1-979-8478609
luhong@tamu.edu

Alessandro Forin
Microsoft Research
One Microsoft Way
Redmond, WA 98052
+1-425-7061841
sandrof@microsoft.com

ABSTRACT

The PSL-to-Verilog (P2V) compiler can translate a set of assertions about a block-structured software program into a hardware design to be executed concurrently with the execution of the software program. The assertions validate the correctness of the software program without altering its temporal behavior in any way, a result that has never been previously achieved by any online model-checking system. The technique and the implementation apply to any general purpose program and the absence of execution overheads makes the system ideal for the verification and debugging of real-time systems.

The assertions are expressed in the simple subset of the Property Specification Language PSL, an IEEE standard originally intended for the behavioral specification of hardware designs. The target execution system is the eMIPS processor, a dynamically self-extensible processor realized with an FPGA. The system can concurrently execute and check multiple programs at a time. Assertions are compiled into eMIPS Extensions, which are loaded by the operating system software into a portion of the FPGA at program loading time, and discarded once the program terminates. If an assertion is violated the program receives an exception, otherwise it executes fully unaware of its verifier. The software program does not need to be modified in any way, it can be compiled separately with full optimizations and executes with or without the corresponding hardware checker.

The P2V compiler is implemented in Python. It generates code for the implementation of the eMIPS processor running on the Xilinx ML401 development board. It is currently used to verify software properties in such areas as testing and debugging, intrusion detection, and the behavioral verification of concurrent and real-time programs.

1. INTRODUCTION

Software program monitoring is an effective approach for the runtime validation of system requirement, usually described in a temporal logic formalism such as LTL formula. In this report, we concern ourselves with transparent monitors, the class of monitors whose execution does not interfere with target programs. Transparent monitoring involves passive observation and online verification. Passive observation refers to the non-intrusive collection of relevant information from an executing program, and online verification refers to the detection of requirement violations in a timely fashion, using the collected information. Most existing observation techniques are based on code instrumentation. Such software based techniques cause the unavoidable probe effect which changes the timing behavior of the target program. Hardware based observation have been attempted in the past with various degrees of success, but to the best of our knowledge, none of them is deployed together with online verification. In this report, we introduce project P2V, a PSL-to-Verilog compilation system, which aims at the runtime verification of real-time as well as general purpose software by automatic generation of the hardware design of a transparent monitor from its sPSL [8] specification.

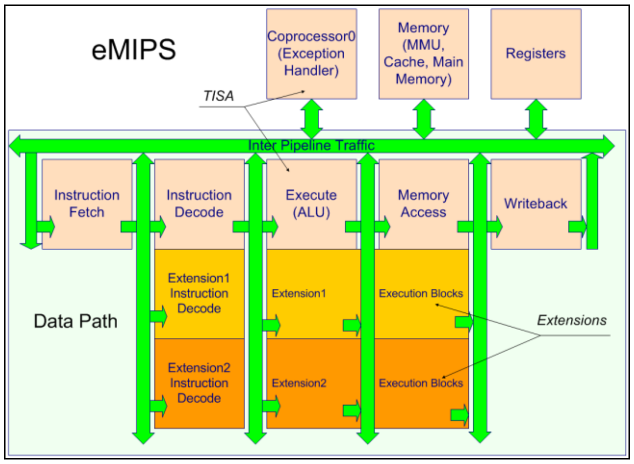

Figure 1: Block diagram of eMIPS architecture

In project P2V, the execution platform for the target software program and its monitor is eMIPS, a dynamically extensible processor [17] implemented on FPGAs. eMIPS allows multiple extensions of a MIPS processor to load dynamically and to plug into the stages of a pipelined CPU data path. Figure 1 illustrates the architecture of the eMIPS platform with two extensions. The transparent monitor unit, abbreviated as MU from this point on, is deployed as one of the extensions. MU has two major components, an observing unit OU and a verification unit VU, as shown in Figure 2. OU is closely integrated with the eMIPS core datapath, and can passively access all relevant signals and registers, including the program counter, the stack pointer, the current instruction register, memory write addresses and values, and the general purpose registers. In Figure 2, this interaction is depicted by the input signals PC, INSTR and MEMVAL among others. VU verifies sPSL assertions using the observations collected by OU. It takes a list of atomic propositions a_1, a_2, …,

a_n as inputs (generated by OU), and outputs two signals VIOLATED and SATISFIED. A concrete example will be given shortly in Section 3 to describe the MU in more details.

Besides being completely transparent, the other distinctive feature of P2V is its flexibility. As an extension to eMIPS core, the logic of MU can be synthesized on a per-program basis. Furthermore, MU is loaded and executed at runtime together with the target program. More than one program can be executing on the same microprocessor under system software control. This flexibility is achieved via the dynamic partial reconfiguration capabilities of modern FPGAs, something that is simply not possible for ASIC platforms.

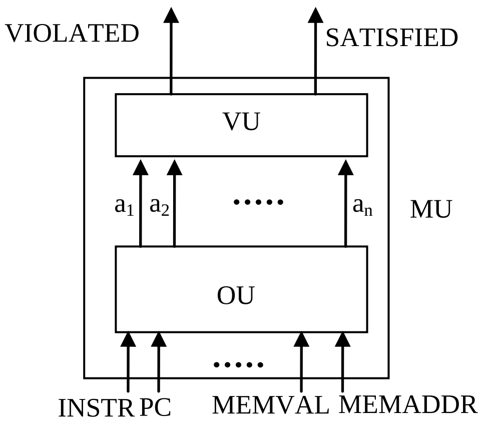

Figure 2: MU architecture

Software correctness specifications are expressed in sPSL [8], a language based on PSL and adapted for C requirement specifications. P2V translates sPSL assertions into Verilog code. The compiled Verilog code is loaded and executed in parallel with the software C program. P2V uses debugging information generated by the C compiler to keep track of the mapping between C and assembly code, and as a result, the binary of the compiled target C program does not need to be instrumented or modified in any way. The dataflow of P2V is shown in Figure 3. The top side of the diagram depicts the normal compilation flow for C, using the standard compiler and tools and resulting in an executable image file. The bottom part of the diagram shows the symmetric flow for the sPSL specifications, compiled by the P2V compiler and resulting into the Verilog source for an eMIPS extension. The manufacturer's FPGA tools then take this file and create the binary file used for partial configuration of the FPGA.

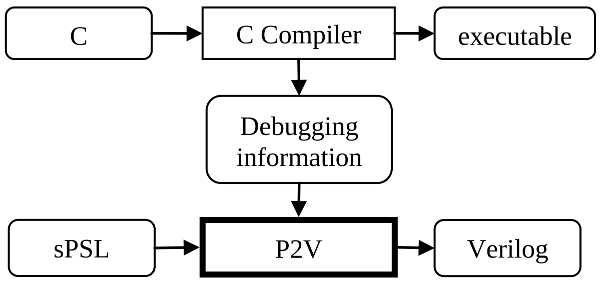

Figure 3: P2V data flow

The remainder of this document is structured as follows. In Section 2 we review the related literature. In Section 3, we use an example to illustrate the design of P2V. A few practical P2V usage cases are presented in Section 4. Section 5 discusses the limitations of our approach, and Section 6 concludes this report.

2. RELATED WORK

Program monitoring has been studied extensively in the past and numerous monitoring systems have been developed and deployed. Existing monitoring approaches can be roughly divided into two groups: software based and hardware based. In this section, we review related work in these two categories.

2.1 Software-Based Monitoring

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 [10][11]. 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 program's source code that is then compiled and executed.

Java-MaC [16] 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.

Roşu [20] 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) [13]. P2V leverages from the work of Roşu and Havelund, it uses their rewriting techniques at compile time to create the monitors, which are then implemented in hardware.

The Java Modeling Language (JML) [15] is a behavioral interface specification language for Java modules. The JML Compiler (jmlc) compiles JML code into runtime checks of the class contracts. In [7], the jmlc compiler is used in conjunction with an Extended Static Checker for Java version2 (ESC/Java2). In [6] 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. The Spec# programming language[5] 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 [3] 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 [4] to perform validation.

All of the above systems insert instrumentation code into the executing program to monitor events and check properties and therefore introduce execution overhead that modifies the program's temporal behavior. This is not acceptable for Real-Time programs and even a limited execution overhead is poorly received by developers.

2.2 Hardware-Based Monitoring

MAMon [12] is a hardware monitoring system that gives non-intrusive observability into the execution of hardware accelerated RealTime Operating Systems. In MAMon, traditional RTOS functions, such as process scheduling, management, and communication are implemented as a hardware unit RTU, whose execution is passively observed by an Integrated Probing Unit IPU, also implemented as hardware. The collected system level events, including task switches, service calls, interrupts, software probes are sent by IPU through a parallel port to a separate host for further processing. MAMon is designed for monitoring the execution of a specific RTOS kernel, while P2V monitors general purpose software at a fine level of granularity.

Noninterference monitoring architecture [21] targets monitoring distributed real-time systems without interfering with their execution by using additional hardware to collect observations of the target system. The data to be captured is predetermined, such as process creation, termination, synchronization, function call and return. Data analysis is performed offline afterwards. Compared with this approach, the type of data to be collected in P2V is much finer grained and dynamically reconfigurable, and data analysis is performed online in a synchronous manner.

ODYSSEY [14] is a system-level synthesis methodology for embedded systems. Recently, an assertion-based verification methodology has been integrated into ODYSSEY. Similar to our approach, specifications of software properties are in PSL and specifications are synthesized into hardware monitors. However, in ODYSSEY, the temporal layer of PSL is only used to specify the validity of a sequence of method calls, while in our system PSL is used to describe the temporal behavior of the entire program, including statements about global and local variables and their values and interactions with other programs and I/O peripherals. Furthermore, only a limited fraction of PSL is supported in ODYSSEY (no until, eventually, and etc), and as a result, the temporal properties that can be specified in ODYSSEY are a very limited subset of those that can be specified in P2V.

```
int ACK = 0;
int control(void)
{     ....
REQ:
       device->CONTROL = 1;
       while(1)
       {
             ACK = device->STATUS;
             ....
        }
}
```

Figure 4: device.c

3. P2V DESIGN

In this section we will use a simple example to illustrate the basic architecture of P2V.

Figure 4 shows a fragment of a simple real-time C program *device.c*. The program acts on the CONTROL of a peripheral device and then loops checking on the STATUS, expecting an acknowledge.

1. atom req := REQ
2. atom ack := ::ACK == 1
3. property P: always(req $\rightarrow$ eventually(ack))

Figure 5: device.c specification

Figure 5 is a fragment of a PSL specification. Line 1 and 2 in Figure 5 define two *atomic proposition* req and ack. The two expressions REQ and ::ACK==1 (used to define req and ack) are called *atomic proposition expressions*. The leading :: before ACK indicates ACK is a global variable. Line 3 defines a *temporal property* P, which asserts that a request must always lead to an acknowledgement. In this example, we explicitly introduced propositions req and ack to specify property P. This is only for ease of later discussion. Otherwise, P can be written more compactly as always(REQ $\rightarrow$ eventually(::ACK == 1)). A complete description of the sPSL syntax and semantics can be found in [8].

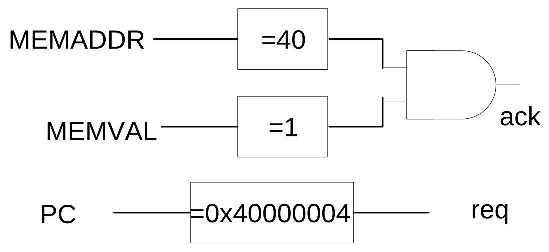

Figure 6: Observation Unit Circuit

3.1 Observation Unit

In the example of Figure 5, the atomic proposition req refers to the label REQ in device.c, and analysis of the compiled program's binary and debugging information reveals that its address is 0x40000004. The atomic proposition ack refers to the variable ACK, which for simplicity we assume is a global variable at address 40 in memory. It might be on the stack, in which case 40 would be the offset from the stack pointer register.

Figure 6 depicts a simplified version of the OU for this example, generated by the P2V compiler. Signals MEMADDR, MEMVAL, and PC come from the eMIPS core datapath. MEMADDR and MEMVAL are compared against 40 and 1 respectively to produce the atomic proposition ack, while the program counter PC is compared with 0x40000004 to produce req. req and ack are later fed to VU for further processing. OU also generates an entry and an exit signals when the entry and exit of function foo are detected. These two signals are omitted in Figure 6 for simplicity.

Based on this simple example, we can now discuss OU in more detail. The main task of OU is execution trace generation. An *execution trace* here is defined as a sequence of *observations*, where an observation is a realization of one atomic proposition. In what follows, we will discuss the two key aspects of execution trace generation: *scope detection* and *atomic proposition evaluation*. Note that the design of OU is inherently platform dependent: OU collects the relevant information about a running program that was originally written in high level programming language by watching the stream of machine code flowing through the CPU. In particular, to generate OU, P2V must have sufficient knowledge of the target platform including the compiler and the hardware architecture. Therefore, the following discussion about OU is based on certain assumptions and conventions of the MIPS platform and its compilers.

In sPSL, the start and the end of a local trace is defined as the entry and the exit of a function invocation. Therefore, function scope detection is one of the main tasks of OU. Assertions can also be scoped globally or limited to a block within a function, but those cases can easily be reduced the local trace case. Let foo be the function of a single threaded C program which OU is observing. Normally, C compilers allocate foo's code at a fixed location in the text segment, and use a stack to organize function invocations. When foo is called, an activation record is created on the top of the stack, which will be discarded when foo returns. By convention, register SP in the MIPS processor is used exclusively to keep track of where the top of stack is. To detect the scope entry of foo, OU simply checks the register PC against foo's virtual address in the text segment. Notice that if foo is recursive it might have multiple activation records on the stack. In this case, the value of register SP at the time when foo is entered is used to distinguish multiple invocations. Furthermore, SP is also used to detect scope exit. For example, suppose the activation record of a particular foo instance is located at memory address 100, then the moment when SP falls below 100 is the moment that that instance goes out of scope. The above simple rule for scope detection can be easily extended to multithreaded programs. In such programs, each thread is associated with a different stack, and as a result, OU only needs to keep track of multiple stacks for correct scope detection.

The primary step for atomic proposition evaluation is to collect the addresses of C program variables, which can be done by analyzing the program binary and debugging information. Typically, C compilers allocate static and global variables at fixed locations in the data segment. Local variables are dynamically allocated on the stack. Function arguments could be allocated either on the stack or in registers. In this report, the process of deciding the location of a variable at the time when it is created (by analyzing debugging information generated by the C compiler) is called *variable analysis*. Variable analysis allows OU to keep track of variables transparently by intercepting and caching all the value written to the variable's location.

3.2 Verification Unit

The verification unit for the property P: always(req → eventually(ack)) from Figure 5 implements the Moore finite state machine (FSM) shown in Figure 7. For this reason, the terms VU and verification FSM are used interchangeably in this subsection. VU has four states INIT, REQ SEEN, SATISFIED, VIOLATED, and is driven by signal req, ack, and exit outputted by OU (the signal entry is omitted for simplicity of illustration). The label of each transition in the verification FSM indicates the values of the observations that trigger this transition. For example, label 100 of transition INIT → REQ SEEN is a non-exit transition triggered by the observation req=1, ack=0, and exit=0. Label **1 of transition INIT → SATISFIED corresponds to an exit observation where the value of req and ack could be either 1 or 0.

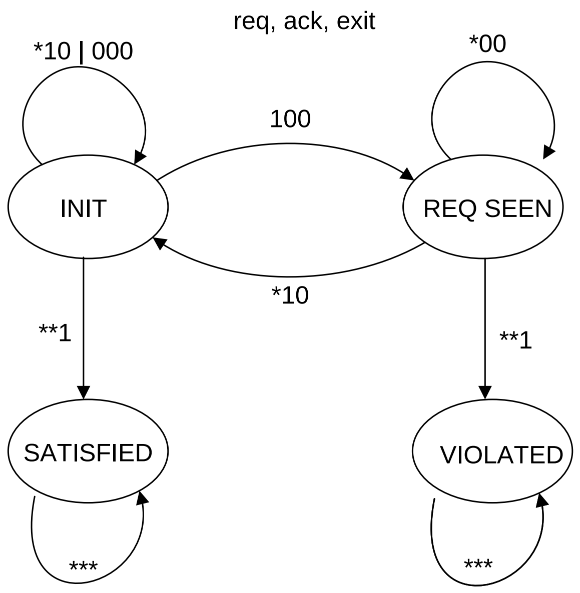

Figure 7: Verification FSM

Driven by the observations generated from the OU, VU verifies the property P defined in example.psl. The moment when it moves to the states SATISFIED/VIOLATED is the moment P is violated/satisfied. For example, suppose VU is at state INIT, and the following trace is observed (by OU):

$O_1 = 100 \quad O_2 = 010 \quad O_3 = 100 \quad O_4 = 001$

Driven by this trace, the action taken by VU is:

INIT→REQ SEEN→INIT→REQ SEEN→VIOLATED

and VU correctly stops at state VIOLATED because the last request was not followed by an acknowledge. It is easy to verify that according to the PSL semantic rules P indeed is violated by this trace. The fact that property violation detection is performed by an FSM is crucial for online verification because the execution of the FSM is completely synchronized with the target program, and the target program needs not be stalled.

The verification FSM in Figure 7 is generated by a property rewriting algorithm proposed by Roşu and Haveland in [20]. The basic idea of this algorithm is that to verify if P holds for a trace starting with an observation O, one just needs to verify if P{O} holds for the rest of the trace, where P{O} is the property rewritten from P given O.

For example, consider the operator **always**. Notice that saying that the property P = always Q holds for a trace starting with O it is equivalent to saying that (1) Q holds for the same trace and, (2) P holds for the rest of the trace.

Therefore, the recursive rewriting rule for an **always** property P is:

$$P\{O\} = Q\{O\} \wedge P.$$

However, if O is an exit observation, then according to the semantics of **always**, P = always Q is satisfied, that is, P{exit} = true. In summary, the rewriting rule for always properties is:

P{O} = P ∨ Q{O},	if O is not exit
P{O} = false,	if O is exit

Given their timeless nature, the rewriting rules for the Boolean operators **and**, **or**, **imply** and **not** are much simpler compared with the temporal operators. Take **and** as an example, P = Q and R holds for a trace starting with O is simply equivalent to saying Q holds and R holds for the same trace, therefore, the rewriting rule for an and property P = Q and R is:

$$P\{O\} = Q\{O\} \wedge R\{O\}$$

The rewriting rules for **or**, **imply** and **not** are similar. The rewriting rules for atomic propositions are straightforward: the property rewritten from proposition A given observation O is simply the value of A in O, which is either true or false. Readers can refer to [21] for a complete list of the rewriting rules.

Now, consider property P: always(req → eventually ack) from our previous example. In what follows, we show how to generate P's verification FSM by property rewriting. P's sub formulae are summarized as follows:

P: always(P_1)	P_1: $P_2 \rightarrow P_3$
P_2: req	P_3: eventually P_4
P_4: ack	

Suppose we want to check if P holds for trace 100,010,100,001, which can be written more clearly as O_1 = req, O_2 = ack, O_3 = req, O_4 = exit given that there is only one true atomic proposition in each observation. P can be verified by the following property rewriting process:

$$\begin{aligned}
P\{O_1\} &= P\{req\} \\
&= P \wedge P_1\{req\} \\
&= P \wedge (P_2\{req\} \rightarrow P_3\{req\}) \\
&= P \wedge (true \rightarrow P_3\{req\}) \\
&= P \wedge P_3\{req\} \\
&= P \wedge (P_3 \vee P_4\{req\}) \\
&= P \wedge (P_3 \vee false) \\
&= P \wedge P_3 \\
P\{O_1\}\{O_2\} &= (P \wedge P_3)\{O_2\} \\
&= P\{ack\} \wedge P_3\{ack\} \\
&= P \wedge P_1\{ack\} \wedge (P_3 \vee P_4\{ack\}) \\
&= P \wedge (P_2\{ack\} \rightarrow P_3\{ack\}) \wedge (P_3 \vee true) \\
&= P \wedge (false \rightarrow P_3\{ack\}) \wedge true \\
&= P \\
P\{O_1\}\{O_2\}\{O_3\} &= P\{O_3\} \\
&= P \wedge P_3 \\
P\{O_1\}\{O_2\}\{O_3\}\{O_4\} &= (P \wedge P_3)\{O_4\} \\
&= P\{exit\} \wedge P_3\{exit\} \\
&= true \wedge false \\
&= false
\end{aligned}$$

The above procedure keeps rewriting properties using new observations. The resulting property $P\{O_1\}\{O_2\}\{O_3\}\{O_4\}$ produced at the end of the trace is false, indicating that the property is violated at that moment. Notice that the above property rewriting chain:

$$P \rightarrow \{O_1\} \rightarrow P \wedge P_3 \rightarrow \{O_2\} \rightarrow P \rightarrow \{O_3\} \rightarrow P \wedge P_3 \rightarrow \{O_4\} \rightarrow false$$

directly corresponds to the state-transition sequence in the FSM in Figure 7:

INIT→REQ SEEN→INIT→REQ SEEN→VIOLATED

where property P corresponds to state INIT, $P \wedge P_3$ corresponds to REQ SEEN, and false corresponds to VIOLATED.

The set of properties that could be generated by rewriting a property is called the *closure* of that property. It may seem that, since there are an infinite number of possible traces and each trace may lead to a new property by property rewriting, there could be infinite number of properties in the closure. It is shown in [20] that the closure of any property is actually finite and its size only depends on the length of the property. The consequence is that the size of the verification FSM is also finite. The algorithm to generate the verification FSM for a property can be summarized as follows:

1. Enumerate all possible observations
2. Compute the closure of the property against all possible sequence of observations.
3. Synthesize the verification FSM such that (a) each state corresponds to a property in the closure, and (b) each transition corresponds to one step in the rewriting rules, according to one observation.

4. APPLICATIONS

In the previous section, we have seen how P2V can monitor one simple liveness property always(req → eventually ack). In this section, we use a few more examples to show how PSL can be used to validate other practical applications.

4.1 Debugging and testing

Consider the C function swap in Figure 8, which takes two pointer arguments x and y, and exchanges the values they point to.

```
void swap(int *x, int *y)
{
    int temp = *x;
L1:
    *x = *y;
    // some other computation
    *y = temp;
L2:
    return;
}
```

Figure 8: swap.c

The requirements that x and y should both be valid and point to two different integers at the entrance of swap can be expressed in sPSL as follows:

$$(x \mathrel{!=} NULL) \wedge (y \mathrel{!=} NULL) \wedge (x \mathrel{!=} y)$$

P2V also allows users to check pointer validity between two specific points in a program. For example, the validity of pointer x between L1 and L2 can be expressed as:

```
always (L1 → (x!=NULL) until L2)
```

P2V can also be used in software testing. As an example, consider a C program with two functions foo and bar. The requirement that function bar must be called immediately after foo at least once in a test suite can be specified as:

```
eventually (foo → next bar)
```

4.2 Real-time behavior verification

The PSL operators next_e and next_a are well suited for specifying real-time properties. Specifically, property next_e [i:j]F holds if F holds for every time instance between i and j from now. Property next_a[i:j] holds if F holds at least once between time i and j from now.

For instance, assume that it is critical in a real-time system that interrupts should never be turned off for more than a short period of time, say 10 microseconds. Assuming that intoff() and inton() are the two functions to turn off and on the interrupts, we can specify this requirement as follows:

```
always( intoff → next_e[0:10µs] inton)
```

next_e, when used together with the operator next_a, is also useful to indicate periodic tasks-- operations that should happen at a given frequency. For example, the requirement that the function foo should be called approximately every 20 milliseconds can be specified as:

```
always( foo → next_a[0:20ms] !foo)
always( foo→ next_e[0:21ms]  foo)
```

4.3 Intrusion Detection

Stack based buffer overflow is a commonly used method to break into computer systems. It usually exploits unbounded string operations to replace the return address of a function on the stack by the address of malicious code. In sPSL, we can detect this kind of attack as follows:

```
never($writing == $return)
```

Here, $writing and $return are two special variables maintained by MU. $writing holds the address of the memory cell that the CPU is currently writing and $return holds the function return address. The proposition $writing == $return evaluates true only when the processor is trying to rewrite the function return address, and the property asserts that this should never happen. In this example, $return is basically a constant. In general, the special variable $writing can also be used to detect attempts to modify program variables that should remain constant. For example, the requirement that variable c is a constant can be specified as follows:

```
never($writing == &c)
```

5. P2V IMPLEMENTATION

The P2V compiler is written in Python. It consists of three major components: a PSL parser, an image parser, and a Verilog code generator. This section describes the implementation of each component, using device.c, swap.c in Figure 4 and Figure 8, and the following demo.psl as example.

```
vunit vunit1(device.c::control)
{
   atom req := REQ;
   atom ack := ::ACK == 1;
   property P := always(req imply eventually(ack));
}
vunit vunit2(swap.c::swap)
{
   atom a := x != 0;
   atom b := x != 0;
   atom c := x != y;
   property P := a and b and c;
}
```

Figure 9: demo.psl

5.1 PSL parser

The PSL parser is implemented in the source files psl.py and property.py located in the folder psllib. The function psl.parse is the entry point of psllib. It takes as input a PSL specification and generates a data structure which is an internal representation of the specification. For example, the result of parsing demo.psl (in Figure 9) is shown in Figure 10.

```
[
  {
        'name': 'vunit1',
        'file_name': 'device.c',
        'func_name': 'control',
        'atoms': {
                'req': 'REQ',
                'ack': ('::ACK', '==', '1'),
        },
        'properties': [
                ('P', ['always',
                        ['imply',
                                'req',
                                ['eventually', 'ack', 0]],
                        1]),]
  },
  {
        'name': 'vunit2',
        'file_name': 'swap.c',
        'func_name': 'swap',
        'atoms': {
                'a': ('x', '!=', '0'),
                'b': ('x', '!=', '0'),
                'c': ('x', '!=', 'y')
        },
        'properties': [
                ('P', ['and', ['and','a', 'b'], 'c']),
        ],
  }
]
```

Figure 10: Output of psllib.parse

The function psl.parse calls psl.vunit_parse, psl.property_parse, and psl.atom_parse to generate the above output. As shown in Figure 10, psllib.parse extracts the name, file_name, func_name, properties, and atoms for each vunit. Each property in properties is an annotated abstract syntax tree generated by property.parse. In this abstract syntax tree, each temporal sub-property of the property is assigned a numerical id. As an example, consider property P of vunit1, the id of its always sub-property is 1, and the id of its until sub-property is 0. The id of a temporal sub-property is used to define the notion of literal. More specifically, a literal is either an atomic proposition or a temporal sub-property id. The notion of literal is crucial to compute the closure of a property. The other useful function in property.py is atoms_of, which computes the sorted list of names of atomic propositions referred to a property.

5.2 Image parser

The image parser is implemented in stab.py located in folder stablib, which is currently still under construction. Stablib uses the output of objdump to obtain the symbol information for all the objects defined in multiple C files, including functions, global variables, local variables, arguments, labels, and so on. The symbol information is stored in a data structure to be consumed by the Verilog generator. For example, stablib.parse generates the following structure for device.c, and swap.c shown in Figure 11.

```
{
    "device.c":{
        # function scope
        "control":{
            "info": (0x80000378, 32, 12),
            "labels": {
                'REQ':0x80000388,
            }
        },
        # global scope
        "":{
            'ACK': (0x800005f4,32,'signed'),
        },
    },
    "swap.c":{
        "swap":{
            "info": (0x80000328, 16, 20),
            "locals": {
                'temp':(-24,32,'signed'),
            },
            "args": {
                'x':(0x10,32,'unsigned'),
                'y':(0x14,32,'unsigned')
            },
            "labels": {
                'L1':0x80000338,
                'L2':0x80000344,
            }
        },
    },
}
```

Figure 11: Output of stablib.parse

More specifically, the extracted information for a function includes function name, address, frame_size, and prologue_size, a locals section, an args (argument) section, an arglocs section, and a labels section. For example, function control in device.c has address 0x80000378, its frame size is 32, and its prologue size is 12. Notice that global variables such as ACK in this example are treated as local variables of a special function with an empty name. This allows us to associate global variables to individual source files. A global variable has name, address, bit width, and sign.

Each local or argument variable has a name, an offset (to the stack pointer), a bit width, and a sign. A label has a name and an address.

5.3 Verilog Code Generator

The Verilog code generator is implemented in the source files P2V.py, MU.py, OU.py, VU.py, and in common_module.py. The entry point of the Verilog code generator is P2V.generate. This function takes as input the output of psllib.parse and stablib.parse, and compiles them into Verilog code. P2V.generate calls to MU.generate, VU.generate, and OU.generate to generate the VU, OU, and MU for each PSL property defined in all PSL vunits. MU.generate is the simplest of the three: it instantiates the OU and VU of a property. Below, we give more details of OU.generate and VU.generate.

5.3.1 OU.py

The function OU.generate calls OU.generate_wires, OU.generate_assigns, and OU.generate_modules to generate the OU for a property given the vunit and stab information. For example, the following Verilog code is generated for the property P of the vunit1 defined in demo.psl, Figure 9.

```
module OU_vunit1_P(
    CLK,PCLK,RESET,GR,PC,
    REGWRITE1_RG,REGWRITE2_RG,
    WRREG1_RG,WRREG2_RG,
    WRDATA1_RG,WRDATA2_RG,
    ADDR_MEM,DATA_MEM,WE_MEM,
    SCOPE,OBSERVATION);

    input CLK, PCLK, RESET, GR;
    input REGWRITE1_RG, REGWRITE2_RG;
    input [4:0] WRREG1_RG, WRREG2_RG;
    input [31:0] WRDATA1_RG, WRDATA2_RG;
    input [31:0] PC;
    input [31:0] DATA_MEM,ADDR_MEM;
    input WE_MEM;

    output SCOPE;
    output [2:0] OBSERVATION;

    wire [31:0] SP,sp;
    wire RANGE, PROLOGUE;

    //wires
    wire atom_ack,atom_req;
    wire label_REQ;
    wire signed [31:0] global_ACK;

    //assigns
    assign OBSERVATION = {atom_ack,atom_req};
    assign atom_ack = (global_ACK == 1);
    assign atom_req = label_REQ;

    //modules
    register_observer #(29) so(PCLK,GR,
       WRREG1_RG, WRDATA1_RG,
       WRREG2_RG,WRDATA2_RG,SP);
    scope_detector #('h80000378,32,12) sd(PC,SP,
       RANGE,PROLOGUE,SCOPE,sp);
    label_observer #('h80000388) lbo_REQ(CLK,
       PC,SCOPE,label_REQ);
    global_observer  #('h800005f4,32) go_ACK(
       DATA_MEM,ADDR_MEM,WE_MEM,global_ACK);

endmodule
```

Figure 12: Output of OU.generate

The value of the global variable ACK is captured by the Verilog modules global_observer go_ACK. The value of register SP is captured by module register_observer so. Scope detection for function control is done by the module scope_detector sd. label_observer lbo_REQ keeps track of register PC (the program counter) and matches it with the address of label REQ. Notice that, the Verilog modules global_observer, register_observer, label_observer and scope_detector are parameterized. In other words, they are designed to be generic modules for any labels, variables, and functions. The definition of these Verilog modules, as well as other common modules and AU are located in common_module.py. AU stands for activation unit. It asks permissions from the eMIPS pipleline arbiter to perform snooping of various TISA signals, such as the PC and SP registers above.

5.3.2 VU.py

VU.generate first calls VU.FSM (which calls VU.closure_of, and VU.rewrite) and VU.FSM_terminal(which calls VU.closure_of and VU.rewrite_terminal) to compute the main and terminal section of the verification FSM of a property, and then uses the result of these two functions to generate the corresponding Verilog code. For example, the Verilog code generated for property P of vunit1 defined in is as follows:

```
module VU_vunit1_P(CLK,SCOPE,OBSERVATION,
    SATISFIED,VIOLATED);
    input CLK;
    input SCOPE;
    // OBSERVATION = ack,req
    input [1:0] OBSERVATION;

    output SATISFIED,VIOLATED;
    reg SATISFIED,VIOLATED;
    reg [20:0] STATE;

    always @(posedge SCOPE)
    begin
       STATE = 0;
       SATISFIED = 0;
       VIOLATED = 0;
    end

    always @(negedge SCOPE)
       case (STATE)
          0: STATE=1;
          1: STATE=1;
          2: STATE=2;
          3: STATE=2;
       endcase

    always @(posedge CLK)
       if (SCOPE)
          case({STATE,OBSERVATION})
             {21'd0,2'b11}: STATE=0;
             {21'd0,2'b01}: STATE=3;
             {21'd0,2'b10}: STATE=0;
             {21'd0,2'b00}: STATE=0;
             {21'd3,2'b11}: STATE=0;
             {21'd3,2'b01}: STATE=3;
             {21'd3,2'b10}: STATE=0;
             {21'd3,2'b00}: STATE=3;
          endcase

    always @(posedge CLK)
       if (STATE == 1) SATISFIED = 1;

    always @(posedge CLK)
       if (STATE == 2) VIOLATED = 1;

endmodule
```

Figure 13: Output of VU.generate

As shown in Figure 13, state 0, 1, and 2 correspond to INIT, SATISFIED and VIOLATED respectively.

5.4 TO-DO list

A proper implementation of an image parser is currently missing.

The implementation of the PSL parser has certain limitations. In the first place, C expressions are not allowed in a property definition. For example, let p be a C integer pointer, then always (p!=NULL) is not a valid property definition. Instead, it should be split into atoms and written as:

atom a := p != NULL

property P: always a

In the second place, at most one C operator is allowed in an atomic proposition definition, and it must be a comparison operator. For example, the following atomic proposition definition is invalid:

atom a := (i == j + k)

The Verilog code generated by P2V has only been tested in a Verilog simulator. It has been successfully run in ModelSim + Giano, but not yet on the FPGA board.

6. LIMITATIONS

In this section, we discuss a few limitations with regard to transparent monitoring and the expressiveness of sPSL.

6.1 Transparency

Transparent monitoring cannot be achieved without placing restrictions on the atomic proposition expressions. Consider the relatively extreme case where a program generates a sequence of numbers, and its monitor must verify that each generated number is prime. It is well known that primarity testing is computational expensive, thus the target program may be generating numbers much faster than its online monitor can consume. As a result, the program needs to wait for its monitor, which results in a change of the timing behavior. Note that this limitation is an inherent one which applies not only to P2V, but also to all transparent monitoring system.

Atomic proposition expressions involving pointer dereferencing may also affect the monitor transparency. Consider an atomic proposition expression **p, where p is a two level C pointer. The value of this expression changes when the value of each level of indirection changes. In general, to keep track of the value of an N-level pointer dereferencing expression we can set a watch address for each level of indirection. At level 0 it is the final object pointed to, at level 1 a pointer, at level 2 a pointer-to-pointer and so on. All transactions to level 0 memory can be transparently intercepted by MU. If level 1 is changed we re-fetch the level 0 value and re-evaluate. And so on for the other levels, at level N we need to dereference N pointers, the object, and re-evaluate. However, the above process must stop the pipeline during these fetches and that changes the program's timing.

For the above reasons, we only allow relative simple computations in atomic proposition expressions, such as equality testing, addition/subtraction. Supported data types in atomic proposition expressions are limited to machine native data types such as byte, word etc.

6.2 sPSL Expressiveness

sPSL is designed for procedural languages. Various issues need to be considered if we want to extend it to object oriented languages. For instance, consider the following C++ code:

```
class foo {
    public:
        void change_something(void);
    private:
        int A, B;
}
```

Figure 14: C++ Example

and the corresponding sPSL property

always(A < B).

This property is what we might call a class level property. Unlike function level properties, it asserts that A must be less than B for every instance of class foo at all times. In this case, variables are associated with an object instance instead of a function and the definition of sPSL semantics has to be adjusted accordingly.

7. CONCLUSIONS

We have introduced project P2V, a PSL-to-Verilog compilation system aimed at realizing an online, zero-overhead verification system both for general purpose and for real-time software. We use Assertion-Based Verification to check the properties of a software program that are expressed in a C binding for PSL, an IEEE standard property specification language. The temporal logic behind PSL is Linear Temporal Logic, which is amenable to online verification. In general, the basic principle of our approach is applicable not only to software written in C, but also other block structured languages.

8. REFERENCES

[1] Accellera and I. 1364, *SystemVerilog*. Accellera Organization, Napa, CA.

[2] Accellera, *IEEE P1850 PSL*. Accellera Organization, Napa, CA.

[3] Ball, T., and S. K. Rajamani, S., K. *SLIC: A Specification Language for Interface Checking (of C).* Technical Report MSR-TR-2001-21, Microsoft Research, Redmond, WA, 2001.

[4] Ball, T., and S. K. Rajamani, S., K. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th SIGPLAN-SIGACT symposium on Principle Of Programming Languages (POPL'02)* (Portland, Oregon, January 16-18, 2002). ACM Press, New York, NY, 2002, 1-3.

[5] Barnett, M., Leino, K., R., M., and Schulte, W. The Spec# Programming System: An Overview. In *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)* (Marseille, France, March 10-13, 2004). LNCS Volume 3362, Springer-Verlag, Berlin, Germany, 2005, 46-69.

[6] Chalin, P., Hurlin, C., and Kiniry, J. Integrating Static Checking and Interactive Verification: Supporting Multiple Theories and Provers in Verification. In *Proceedings of the International Conference on Verified Software: Theories, Tools, Experiments (VSTTE'05)* (Zurich, Switzerland, October 10-13, 2005).

[7] Chalin, P., and James, P. Cross-Verification of JML Tools: An ESC/Java2 Case Study. Technical Report MSR-TR-2006-117, Microsoft Research, Redmond, WA, 2006.

[8] Cheung, P. H., and Forin, A. A C-language binding for PSL. In *Proceedings of the 3rd International Conference on Embedded Software and Systems (ICESS'07)* (Daegu, Korea, May 14-16, 2007). LNCS Volume 4523, Springer-Verlag, Berlin, Germany, 2007, 585-591.

[9] Curcio, I., D., D. A Simple Assertion Pre-processor. *ACM SIGPLAN Notices, 33* (December 1998), 44-51.

[10] Drusinsky, D. The Temporal Rover and the ATG Rover. In *Proceedings of the 7th SPIN Workshop on Model Checking and Software Verification (SPIN'00)* (Stanford, CA, August 30-31, 2000). LNCS Volume 1885, Springer-Verlag, Berlin, Germany, 2000, 323-330.

[11] Drusinsky, D. Monitoring Temporal Rules Combined with Time Series. In *Proceedings of the 15th Computer Aided Verification Conference (CAV'03).*(Boulder, CA, July 8-12, 2003). LNCS Volume 2725, Springer-Verlag, Berlin, Germany, 2003, 114-118.

[12] El Shobaki, M. On-Chip Monitoring of Single-and Multiprocessor Hardware Real-Time Operating Systems. In *Proceedings of the 8th International Conference on Real-Time Computing Systems and Applications (RTCSA'02).* (Tokyo, Japan, March 18-20, 2002).

[13] Havelund, K., and Roşu, G. Java PathExplorer --- A runtime verification tool. In *Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics and*

Automation in Space(ISAIRAS'01). (Montreal, Canada, June 18-20, 2001).

[14] Hessabi, S., Gharehbaghi, A., M., Yaran, B., H., and Goudarzi, M. Integrating assertion-based verification into system-level synthesis methodology. In *Proceedings of the 16th International Conference on Microelectronics(ICM 2004)* (Tunis, Tunisia, December 6-8, 2004). IEEE Press, Catalog 04EX918, New Brunswick, NJ, 2004, 232-235.

[15] Leavens, G., T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Muller, P., Kiniry, J., and Chalin, P. *JML Reference Manual*. Iowa State University, Ames, IA, 2006.

[16] Lee, I., Kannan, S., Kim, M., Sokolsky, O., and Viswanathan, M. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications(PDPTA'99)* (Las Vegas, NV, June 28-30, 1999). CSREA Press, ISBN 1-892512-15-7, 279-287, 1999.

[17] Pittman, R., N., Lynch, N., L., and Forin, A. *eMIPS, A Dynamically Extensible Processor.* Technical Report MSR-TR-2006-143, Microsoft Research, Redmond, WA, 2006.

[18] Pnueli, A. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)* (Providence, RI, October 31-November 2, 1977). IEEE Press, New Brunswick, NJ, 1977, 46-57.

[19] Prior, A., N. *Past, Present and Future*. Oxford University Press, Oxford, UK, 1967.

[20] Roşu, G., and Havelund, K. Rewriting-based Techniques for Runtime Verification. In *Proceedings of the 16th IEEE Conference on Automated Software Engineering (ASE'01)*(Coronado Island, CA, November 26-29, 2005). IEEE Press, New Brunswick, NJ, 2005, 135-143.

[21] Tsai, J., J., P., Fang, K.-Y, Chen, H.-Y., Bi, Y.-D. A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging. *IEEE Transaction on Software Engineering*, 16:8 (August 1990), 897-916.