Assertion-Based Verification through Binary Instrumentation

Enzo Brignon, Laurence Pierre
Univ. Grenoble Alpes, CNRS, Grenoble INP, TIMA
38000 Grenoble, France

***Abstract*—Verifying the correctness and the reliability of C or C++ embedded software is a crucial issue. To alleviate this verification process, we advocate runtime assertion-based verification of formal properties. Such logic and temporal properties can be specified using the IEEE standard PSL (Property Specification Language) and automatically translated into software assertion checkers. A major issue is the instrumentation of the embedded program so that those assertion checkers will be triggered upon specific events during execution. This paper presents an automatic instrumentation solution for object files, which enables such an event-driven property evaluation. It also reports experimental results for different kinds of applications and properties.**

I. Introduction

Since cutting-edge embedded systems include increasingly sophisticated components, the demand for efficient verification solutions is growing. We advocate the use of *runtime Assertion-Based Verification* (ABV). It aims at checking whether applications obey properties, usually expressed as *temporal* formulas (assertions) that capture the design intent. These properties are transformed into assertion checkers, triggered during execution to report property satisfaction or violation. Temporal assertions can be used for debugging purposes; they can also be useful for the online monitoring of fault-tolerance, security, or performance properties.

Languages like the IEEE standards SVA (SystemVerilog Assertions) and PSL (Property Specification Language) [1] are typically used to formalize temporal assertions [2]. When used for the expression of requirements about *hardware* systems, temporal assertions usually express expected properties for *synchronous* hardware components, at the bit level. Thus they are simply checked on clock ticks (in other words, the execution trace is sampled on clock ticks).

In contrast, the work presented here applies ABV to *software* components, for requirements about the software behaviour. *Events* that will trigger checker's evaluations should be updatings and readings of the program variables (both global and local), and function calls or returns. The contribution of this paper is the specification and the implementation in our OSIRIS tool of two solutions to automatically instrument object files of embedded programs in order to enable such an event-driven activation of temporal assertion checkers. Experiments have been performed on a Raspberry Pi (Cortex-A7) and on a Zybo board (dual-core Cortex-A9). We report experimental results (on the Raspberry Pi) for three use cases: a simple software and two realistic embedded applications.

II. Brief State of the Art

Various approaches for monitoring embedded systems are described in [3]. Hardware monitoring is not intrusive in the target software. Software monitoring, which is more flexible, mainly uses source code instrumentation, modifications of the operating system, or process monitoring. Most software instrumentation solutions target event-triggered monitoring.

A. Event-triggered monitoring

Thaker [4] proposed an ABV solution for Java programs that uses aspect-oriented programming to instrument source code. Linear Temporal Logic (LTL) assertions can be specified in terms of methods calls/returns and global variables read/write actions. The temporal logic considered in our tool is comparable to LTL, but we also enable conditions on local variables and function parameters.

The Temporal Rover tool [5] targets LTL extended with Metric Temporal Logic (MTL) assertion-based verification. This tool instruments source code and the instrumentation uses an external software that runs on a host machine. Properties are located inside functions, they involve changes of values of variables in the scope of the function.

The RMOR framework [6] is also an event-triggered verification tool, but it generates monitors for properties specified as Finite State Machines (FSM), which correspond to the expected behavior of the targeted application. It uses an aspect-oriented programming language to instrument the source code.

In [7], we proposed two event-triggered solutions to perform ABV for PSL assertions. The first one modifies the source code, and the second one is a *ptrace*-based process monitoring method for dynamic binary instrumentation. The latter induces a prohibitive CPU time overhead. The object file instrumentation techniques proposed here are much more efficient.

B. Some other approaches

The RiTHM tool [8] adopts the Time-Triggered Runtime Verification (TTRV) model i.e., the monitors are invoked periodically using a predefined frequency, to evaluate LTL properties (only over global variables). This solution has a fixed time overhead, it is generally more time-consuming than ETRV (Event-Triggered Runtime Verification).

Finally, [9] gives a C-language binding for PSL that enables the simulation-based verification of PSL properties. It uses a hardware/software simulator to perform analysis of the events involved in the properties.

978-3-9819263-2-3/DATE19/©2019 EDAA

III. Embedded Programs and Temporal Assertions

PSL assertions can be associated with C programs to *debug* the algorithm itself or its implementation. Even though also reported in section V, CPU time overhead is not crucial in that case. A second application can be the online (embedded) monitoring of *fault-tolerance* or security properties. Finally, such property checkers can also serve as online *performance* analysis components, as demonstrated in the third use case. In those two contexts, CPU time overhead is more relevant.

A. Use case 1: Optimization by simulated annealing

This first use case [10] is not an embedded application, but it provides a simple illustration of the use of temporal properties for debugging purposes. Simulated annealing starts from an initial solution s_0 and iterates by visiting neighbouring solutions. If the new solution `new_E` is better than the previous one `E`, this solution is accepted. If it is worse, a function `boltzmann` is used to decide whether the algorithm shall accept it and step forward, or choose another neighbour. Some expected properties (already proposed in [7]) are:

- (P_1_S) Everytime a new solution with an energy `new_E` lower than the energy of the best solution `best_E` is found, then `best_E` will be updated with this `new_E`
- (P_2_S) In addition to the first property, if `best_E` is updated, it will not be modified again (no alteration) *until* `new_E` is recomputed
- (P_3_S) Everytime the algorithm finds a better solution than the previous one (i.e., `new_E` $<$ `E`), this solution will unconditionnaly get accepted (i.e., the `boltzmann` function will not be called until `new_E` is recomputed)
- (P_4_S) Everytime the algorithm finds a worse solution than the previous one, the `boltzmann` function must be called *before* determining the next solution.

B. Use case 2: Path-following controller

The second use case is an embedded application that implements a path-following lateral controller for an autonomous car [11]. Sensors below the car detect the location of the desired path. The values captured by these sensors are analyzed by a DSP (Digital Signal Processor) which sends to the controller a velocity, and an angle that is used to determine how to turn the steering wheels. To transmit the velocity or the angle to the controller, it copies the value to a variable `output`, then it calls a function `putMem`. The properties below might be considered during debug:

- (P_1_C) and (P_2_C) If the front sensors detect the path on the left (resp. right), then the next assignment of an angle to `output` will be such that a correction is applied i.e., the difference between the previous value of the angle in `output` and the current one is positive (resp. negative),
- (P_3_C) and (P_4_C) If the front sensors detect the path on the left (resp. on the right) and then do not detect it anymore, then the next assignments of angles to `output` will be such that a correction is applied *until* the front sensors detect the path again.

Some other properties can participate in *fault-tolerance* monitoring e.g., w.r.t. electromagnetic perturbations. For example, violations of the properties below would detect that, once a value is computed for the angle, it further changes before being sent to the microcontroller:

- (P_5_C) Every *last* assignment to `angle` *before* an assignment to `output` actually corresponds to the value that is written into `output`,
- (P_6_C) When the algorithm is in the process of computing an `angle` (not a velocity), then a flag in `output` is set to indicate an angle, and the *next call* to `putMem` will actually receive this value of `output` as parameter.

C. Use case 3: Audio decoding

This use case is the software executed on an audio decoding system that mainly includes a processor, memories, a PMU (power management unit), and a DAC for the audio output. The software gets frames from the audio files, decodes them, and writes them to the buffer of the audio out. When this buffer is full, a call to `wait_for_interrupt` enables to stop the processor clock until any notification that the buffer is not full anymore. More precisely, the clock is stopped until the next hardware interrupt. If the interrupt does not come from the DAC, then `wait_for_interrupt` should be called again. Some properties typical of the expected behaviour are:

- (P_1_M) Everytime the buffer is full, function `wait_for_interrupt` will be called, and will be called again *until* receipt of the DAC interrupt
- (P_2_M) Everytime the buffer is full, no write attempt will occur i.e., function `write_buffer` will not be called, *until* receipt of the DAC interrupt. For example, it can be formalized in PSL for OSIRIS as follows:

```
ALWAYS (buffer_full#END() && buffer_full.p#0 == 1
   => !write_buffer#START()
      until! (wait_for_interrupt#END() &&
            wait_for_interrupt.p#0 == INTNOTFULL)));
```

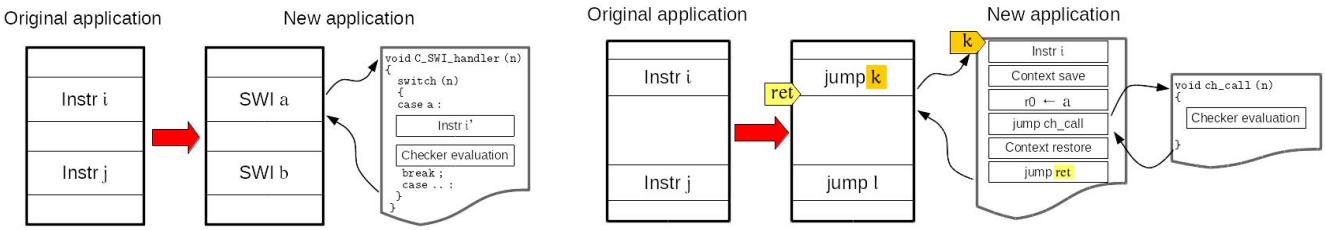
(where `f#START()` denotes that function `f` is starting its execution, `f#END()` denotes that function `f` has just returned, and `f.p#0` denotes its return value).

Our solution also enables to realize *performance* monitoring:

- We consider property (P_3_M) Every frame is fully processed in less than 5 ms (i.e., the time elapsed between the beginning and the end of each processing is $<$ 5 ms)
- We can also use the instrumentation solutions of section IV to trigger other functionalities than PSL assertions: they trigger counter increments instead, in order to perform *profiling*. We use this feature here to verify that decoding a frame is never longer than reading it.

IV. Instrumentation for Software Monitoring

PSL assertions are translated into C++ assertion checkers [12], and the mechanism that enables their event-triggered activation is inspired from the Observer pattern [13]. The events of interest are (1) global/local variable readings and updatings, and (2) function calls and returns. The PSL parser



(a) Through SWI call

(b) With jump to checker

Fig. 1: Binary instrumentation solutions

of our OSIRIS tool recognizes them in the assertions, for example `write_buffer#START()` in the assertion above. Then OSIRIS identifies in the object files the addresses of the corresponding instructions (ARM load/store instructions in the former case, and data processing and branching instructions in the latter). To locate them, OSIRIS uses data included in the relocation and symbol tables (.rel.text and .symtab) and debugging information contained in DWARF sections.

Finally OSIRIS automatically produces a new binary file that integrates the original application instrumented with the assertion checkers through the observation mechanism intended to trigger them on the observed events.

A. Checker activation through software interrupts

We first implemented a solution well-adapted to bare metal contexts, in which the interrupt vector table (IVT) can easily be modified with new interrupt handler addresses. Its principle is depicted by Fig. 1(a) in which we assume that instructions i and j induce events that must trigger a checker evaluation. They are automatically replaced by a software interrupt instruction `SWI`. Its parameter is a number that identifies the replaced instruction, for example instruction i becomes `SWI` a. Upon the execution of this software interrupt, control will be passed to a specific interrupt handler, `C_SWI_handler`. It must execute the replaced instruction and trigger the checker evaluation. The form of this handler function is given by Fig. 2. It receives as parameters the interrupt number of the `SWI` instruction ($number$), the address on the stack from which the calling context has been saved (reg), and the values of the frame pointer (fp) and $r0$ registers of the previous context. Those two parameters are used to retrieve the values of parameters/local variables and return values (stored in $r0$).

```
void C_SWI_handler (int number, unsigned* reg,
                    int fp, int r0) {
  switch (number) {
    // ...
  case a:
  { register unsigned* stack asm("r0")= reg;
    __asm ("ldr r2, [%0, #8]" : "=r" (stack));
    __asm ("ldr r3, [r2, #20]");
    __asm ("str r3, [%0, #4]" : "=r" (stack));
    update_checkers(number, fp, r0);
    break;
  }
    // ...
  }
}
```

Fig. 2: Form of the C_SWI_handler function

In each alternative of the switch statement, the first lines are assembly instructions devoted to the execution of the original instruction, and the last one triggers the checker (`update_checkers`). In the example `case a:` of Fig. 2, the replaced instruction i is supposed to be `ldr r0, [r1, #20]`. The generated substitute instructions fetch the original context into registers in order to execute i within its new context (i.e., in the handler function). The original `ldr r0, [r1, #20]` loads the value pointed to by register $r1$ plus an offset of 20 into register $r0$. Its substitution i' is :

- the value of reg (top of the stack) is loaded into $r0$
- the value of $r1$ is fetched from this stack and loaded into a general-purpose register, for example $r2$
- the value of $r2$ plus an offset of 20 is loaded into another general-purpose register, for example $r3$
- finally the value of $r3$ is stored on the stack, at the location where $r0$ has been saved.

In addition to inserting the `SWI` instructions in the object file, OSIRIS can automatically generate this handler function with the corresponding substitute code, for every type of instruction.

Note that a new interrupt vector is also inserted in the IVT: first it pushes registers $r14$ (lr) and $r12$ to $r0$ on the stack, then it calls `C_SWI_handler`.

B. Jump to checker evaluation

Since it requires to modify the IVT, the previous solution is not manageable with an OS. Thus a more general solution directly replaces instructions by branch instructions to the function that triggers the checker evaluation, called `ch_call` below. Its principle is described by Fig.1(b), where a is a number that identifies the replaced instruction i. The parameter k of the substitute branch instruction (here referred to as $jump$) is a symbol that labels an additional piece of code that redirects to function `ch_call`. With this solution, no interrupt mechanism saves/restores the context before/after passing control to `ch_call`. Therefore instruction $jump\ k$ must branch to additional assembly instructions that:

- execute the replaced instruction i (same context here)
- save the context before calling `ch_call`
- call the `ch_call` function, it receives a as parameter
- restore the context
- go back after the replaced instruction. To identify the corresponding location, a specific symbol (ret on the picture) has automatically been added in the symbol table.

Use case and properties	CPU time without properties (seconds)	CPU time with monitoring (seconds)		Activity of the checkers	
		With properties	Instrumentation only	Activations	Evaluations
Simulated annealing (TSP)	16.79				
(P_1_S)		23.78	17.63	9225007	23
(P_2_S)		24.16	17.54	9225007	5007
(P_3_S)		31.78	18.19	19944582	1505190
(P_4_S)		33.41	18.14	19944582	7714810
All properties		56.78	18.27		
Car controller	18.33				
(P_1_C) & (P_2_C)		19.13 (+ 4.3%)	18.24	1322640	202905
(P_3_C) & (P_4_C)		23.09 (+ 25.9%)	18.40	1653302	83166
(P_5_C) & (P_6_C)		20.67 (+ 12.7%)	18.08	1653302	330662
All properties		26.52	18.66		
Audio decoding platform	120.08				
(P_1_M)		120.66 (+ 0.4%)	120.63	10392342	1803
(P_2_M)		120.74 (+ 0.5%)	120.32	5199975	1869
(P_3_M)		120.65 (+ 0.4%)	120.35	6740	2231
All properties		120.94	120.86		

TABLE I: Experimental results on a Raspberry Pi (ARM Cortex-A7) under Linux

Since instruction i is executed before calling `ch_call`, this function neither needs to execute it, nor to include a switch statement. It only triggers the checker evaluation (`update_checkers`).

OSIRIS automatically inserts the $jump$ instructions in the object file and performs some modifications (see below), generates the `new_code.s` file that contains the assembly code for the actions described above, as well as the `ch_call` C function. Both resulting object files will be linked with the object file of the instrumented application, modified as follows:

- the symbols k and ret are inserted in the symbol table. The value of ret is the address of the instruction that follows the replaced instruction, and the value of k is 0 because k is still unknown (resolved at linking time)
- a new entry is inserted in the relocation table, it correlates the address of the branch instruction $jump\ k$ and k.

The linker will use these new table entries to resolve the addresses. It merges the symbol tables of all the object files, and associates k with the address of the first instruction of the file `new_code.s`. On the other hand, it uses the new information in the relocation table to set the target address of the *jump* instruction to the value of k.

V. Experimental Results and Conclusions

We report experiments (summarized in Table I) with the solution of section IV-B, which is the most widely usable. Column 1 gives CPU times for raw executions, and column 2 gives CPU times with property monitoring. To emphasize the overhead induced by the instrumentation itself, column 3 gives the corresponding CPU times i.e., without the checker evaluations. Columns 4 and 5 give the total number of checkers activations (notifications on events), and of checkers actual evaluations (the condition of the implication premise holds).

For simulated annealing, CPU time is not crucial since all properties target debugging activities. However, we can observe that the overhead which is purely due to instrumentation is about 8%, in spite of the large amount of activations.

For the path-following lateral controller, the experiments are for about 330000 acquisitions of sensor values. Pure instrumentation overhead is negligible. The overhead for properties (P_5_C) and (P_6_C) is about 12.7%, which suggests that these checkers could actually be used for online monitoring.

The experiments for the audio decoding application correspond to decoding a two-minute track. The overhead for checking all the properties is negligible. Additionaly, we have used the instrumentation to profile functions `frame_decode` and `frame_input`, thus verifying that decoding a frame is never longer than reading it: decoding requires an average of 14 ns, whereas reading takes an average of 372 ns.

Note that, on similar use cases, time overhead has the same order of magnitude as with source code instrumentation [7].

References

[1] PSL, *IEEE Std 1850-2010, IEEE Standard for Property Specification Language*. IEEE, 2010.

[2] J. Havlicek and Y. Wolfsthal, "PSL and SVA: Two Standard Assertion Languages Addressing Complementary Engineering Needs," in *Proc. DVCon'2005*, Feb. 2005.

[3] C. Watterson and D. Heffernan, "Runtime verification and monitoring of embedded systems," *IET Software*, vol. 1, no. 5, October 2007.

[4] S. Thaker, "Runtime monitoring temporal property specification through code assertions," in *Department of Computer Science, University of Texas at Austin*, 2005.

[5] D. Drusinsky and M. Shing, "Monitoring temporal logic specifications combined with time series constraints," *Journal of Universal Computer Science*, vol. 9, no. 11, November 2003.

[6] K. Havelund, "Runtime Verification of C Programs," in *Proc. TestCom'2008*. Springer-Verlag (LNCS 5047), 2008.

[7] M. Chabot, K. Mazet, and L. Pierre, "Automatic and Configurable Instrumentation of C Programs with Temporal Assertion Checkers," in *Proc. ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, September 2015.

[8] S. Navabpour *et al.*, "RiTHM: A Tool for Enabling Time-triggered Runtime Verification for C Programs," in *Proc. FSE*, St. Petersburg, Russia, 2013.

[9] P. Cheung and A. Forin, "A C-Language Binding for PSL," in *Proc. Third International Conference on Embedded Software and Systems (ICESS)*. Springer-Verlag (LNCS 4523), 2007.

[10] GNU Scientific Library, http://www.gnu.org/software/gsl/.

[11] P. Mellodge, "Feedback Control for a Path Following Robotic Car," Master of Science Virginia Tech, April 2002.

[12] L. Pierre and L. Ferro, "A Tractable and Fast Method for Monitoring SystemC TLM Specifications," *IEEE Transactions on Computers*, vol. 57, no. 10, October 2008.

[13] E. Gamma *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.