Environment Modelling for Tighter Timing Analysis of Synchronous Programs

Siddharta Andalam, Roopak Sinha and Partha S Roop
Precision Timed Systems Research Group
Department of Electrical and Computer Engineering
University of Auckland, New Zealand
Email: sand080@aucklanduni.ac.nz, {r.sinha, p.roop}@auckland.ac.nz

Abstract—Static timing analysis of a hard real-time application is necessary to ensure that task-level timing deadlines are always met. In many cases, it is preferable to include details about the operating environment to ensure precise timing analysis. However, adding an environment model increases the overall state space being analyzed, which can result in longer analysis time.

In this paper, we present three approaches for modelling the environment. The first approach, which uses variables, provides precise timing results but causes state space explosion. This tremendously increases analysis time. The second approach, which uses real-valued clocks, provides an abstraction over variables that helps reduce the analysis time, when we trade-off precision. A third approach, based on a heuristic, uses a combination of both variables and clocks, and can optimize the state space while achieving precise timing results.

Experimental results show that the proposed heuristic achieves a 91% reduction in the state-space as compared to using variables, while maintaining the same precision.

I. Introduction

Embedded applications ranging from complex flight controllers to digital cameras require to satisfy strict timing deadlines, and hence are called real-time systems. Safety for such systems needs to be guaranteed using formal techniques to avoid any possible disasters. This requires systems to be verifiable statically. Synchronous languages [5] are a popular formal model of computation based on the synchrony hypothesis, allowing systems to be easily validated using formal analysis techniques such as model checking.

Synchronous programs execute in a notion of ticks. During each tick, program first samples the environment. Then a reactive function is called to compute/update control and data. Finally it emits the new control and data to the environment, thus ending a tick. Synchrony hypothesis requires that the idealised system reacts faster compared to the changes in the environment. In order to ensure that this hypothesis is satisfied, we need to determine the worst case reaction time (WCRT) of the program, or the maximum time any tick in the program can take to execute.

In contrast to WCRT analysis for synchronous programs, worst case execution time (WCET) analysis determines the worst cases delay path in a conventional program. WCET analysis can be done statically at compile time [14], or by using measurement-based analysis [13]. The focus of this article is on static WCRT analysis of synchronous programs.

In order to carry out effective WCRT analysis, we must have a formal model that closely reflects the execution model of the program being analyzed. In most cases, and especially in the case of real-time embedded systems, this analysis is incomplete without taking the operating environment of a program into account. Different environments may affect the timing behaviour of a system under test (SUT) differently. For example, under a specific environment, some paths in the SUT may be infeasible, and hence must be ignored during the WCRT analysis.

In [11], we presented a technique to carry out WCRT analysis of synchronous programs written using PRET-C [4]. In this paper, we extend this to incorporate an environment model during static analysis to provide tighter results. We model only those aspects of the environment that can affect the SUT’s timing behaviour. In our case, this model is represented by environment inputs modelled as bounded variables, and we carry out WCRT analysis over all possible valuations of every such input. We explore the use of two representations for modelling environment inputs - data variables and clocks, and propose a third alternative using a heuristic that optimizes both WCRT analysis time and accuracy.

A. Related Literature

Environment modelling for the timing analysis of real-time systems has been an active field of research. A number of techniques propose environment models that help achieve more accurate timing analyses for SUTs. Table I lists key approaches and compares them with the technique proposed in this article.

Taxys [6], TTG [9], and aIT [12] are static timing analysis techniques. xGIOTTO [8] is a language for writing real-time systems and its compiler allows some static analysis to check for problems such as race conditions, and timing safety. TWCT [10] is a measurement based timing analysis method that is used to confirm WCET results obtained from other static analysis methods. All of these techniques focus on WCET analysis, whereas the focus of our formulation is WCRT analysis.

All approaches use different models for SUTs. Some approaches like TTG and xGIOTTO use timed automata based models, while some others [10], [12] choose to extract timing information directly from code. The closest approach
to ours in the choice of SUT is Taxys [6], which uses the synchronous language Esterel, along with C, to model SUTs. We use PRET-C, which is a synchronous language based on C.

It is also interesting to note that each technique has a different notion for what constitutes an environment. Taxys and TTG model the operating environment using non-deterministic semantics, and include the hardware platform on which a program executes as part of the environment. In the aIT approach [12], the environment model is used to simplify the system for timing analysis by using process and domain abstraction. xGIOTTO and TWCET [10] use input variable ranges to model the environment, and hence are the closest to our technique in this aspect.

While some techniques like Taxys and xGIOTTO use a logical notion of time (based on the ticks of a logical clock), others (including us) use real execution-time of various parts of the model. These techniques also allow users to specify the timing characteristics of a SUT. xGIOTTO does not allow the user to specify additional specifications, and uses implicit checks in its compiler to carry out timing analysis. In aIT and TWCET, no additional specifications are allowed, and the techniques are employed to simplify the SUT for static timing analysis and verify an existing WCET obtained from static timing analysis respectively. Only Taxys and TTG allow users to specify additional timing specifications. In our case, the specification constitutes an estimate of the WCRT to be verified.

It should also be noted that a plethora of analysis algorithms, ranging from model checking to genetic algorithms, are employed by the various techniques. In terms of comparison, only Taxys employs a model checker to carry out static timing analysis, which is similar to our technique.

Table I provides a comparison of our technique with existing approaches, and helps highlight the main contribution of this article. While the idea of modelling the environment to achieve precise timing analysis is not new, the use of different representations (data variables and clocks) for SUT inputs to reduce WCRT analysis time (without affecting the accuracy of the analysis) is a new concept that has not been explored in earlier.

The key contributions of this paper are summarized as follows:

1) We model the environment for tighter timing analysis.
2) We present a heuristic based on real-time clocks for faster analysis.
3) Through our experiments, we demonstrate that the proposed heuristic maintains tight estimate bounds while reducing the analysis time.

The rest of this paper is organized as follows. In section II we summarize our WCRT analysis approach using a motivating example. In section III we show how environment models are created in our new approach. Implementation results obtained from this new technique are presented in section IV. Finally, concluding remarks appear section V.

II. Background

A. PRET-C

PRET-C [4] is a subset of the C language, extended with the idea of *synchrony* based on synchronous languages [5]. Unlike other synchronous languages, threads in PRET-C have fixed priorities, and by construction it supports safe communication between threads using shared variables.

Fig. 1 presents a PRET-C program. PRET-C macros are included from the pretc.h file. On line 2, user defined functions are included from the foo.h file. Line 3 declares a reactive input sensor1, which is sampled at the start of every tick. It is of type *int*, with a lower bound of 10 and an upper bound of 28. Line 6 declares a reactive output out, which is emitted to the environment at the end of the tick. Line 7 declares a global variables s1 and s2 for shared communication between threads sampler and processor. Thread sampler samples the reactive inputs sensor1 and sensor2. Then a *moving average* of both signals, is calculated and written on to the

<table>
<thead>
<tr>
<th>Technique</th>
<th>Model of SUT</th>
<th>Environment model</th>
<th>Model of Time</th>
<th>Specifications</th>
<th>Timing Analyzer</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>xGIOTTO [8]</td>
<td>Time-triggered with asynchronous events</td>
<td>Implicit using event scoping</td>
<td>Logical time</td>
<td>Implicit (race conditions, memory usage, timing safety)</td>
<td>Compiler</td>
<td>Game-based</td>
</tr>
<tr>
<td>aIT [12]</td>
<td>VHDL</td>
<td>Multiple abstractions</td>
<td>Real-time (instruction-level)</td>
<td>–</td>
<td>aIT</td>
<td>Abstract Interpretation</td>
</tr>
<tr>
<td>Proposed technique</td>
<td>PRET-C</td>
<td>Bounded variables using variables &amp; and clocks</td>
<td>Real-time</td>
<td>Temporal logic</td>
<td>UPPAAL</td>
<td>Model checking</td>
</tr>
</tbody>
</table>

Table I

A QUALITATIVE COMPARISON BETWEEN TIMING ANALYSIS TECHNIQUES USING ENVIRONMENT MODELLING
shared variables s1 and s2 (line 12 and line 30). Thread processor runs a simple algorithm over the shared data (s1 and s2 ) to compute x (line 28). Then, in the next tick, the reactive output out is updated with the value of x. The main thread spawns sampler and processor using the PAR statement on line 37. The textual ordering gives sampler priority over processor. This PAR is nested inside a strong Abort which evaluates the preemption condition reset<=10 (line 38) at the beginning of every tick. If the PAR is preempted, both threads terminate. More details of the PRET-C language appear in [4].

```c
#include <pretc.h>
#include <foo.h>
ReactiveInput(int, sensor1, 10, 28); // lower bound 10
ReactiveInput(int, sensor2, 70, 94); // upper bound 94
ReactiveInput(int, reset, 0, 20);
ReactiveOutput(int, out);
int s1, s2;

void sampler(){
  int s1_avg[5];
  int s2_avg[5];
  while(1){
    s1=calAvg(sensor1, s1_avg);
    s2=calAvg(sensor2, s2_avg);
    EOT;
  }
}

void processor(){
  int x;
  while(1){
    while(s2>0){
      if(s1>s2){
        s1=s1-s2;
        Abort;
      } else{
        s2=s2-s1;
      }
    } when(reset<=10);
    EOT;
    x++;
    out=x;
  }
}

Figure 1. Example to illustrate the effect of environment on timing analysis.

B. Timing Model and WCRT Analysis

To capture the temporal nature of a PRET-C program, we first compile the program using the gcc compiler to obtain the assembly code. Then from the assembly code, we extract the control-flow of the program along with its temporal characteristics (execution time of instructions). This control-flow graph also captures the high level concurrency of the program, and is called a timed concurrent control flow graph (TCCFG). More details on this process are presented in [4].

The TCCFG is converted into a corresponding timed automaton (TA) that can be analyzed using the UPPAAL model checker [1]. Figure 2 shows the abstracted timing model of the processor thread, presented in Fig. 1. Transitions between nodes have associated costs which reflect the number of clock cycles needed to execute the corresponding part in the actual program. WCRT is calculated by incrementing a variable called WCRT between transitions.

The WCRT value provided by this approach is usually an over-estimate of the actual value. One of the reasons for this over-estimation is the fact, we did not take the operating environment of the program into account during the earlier WCRT analysis [11]. Multiple outgoing transitions from a node that were conditional to different values of an environment input will all be considered valid, if no information about the environment is available. This can lead to an overestimated WCRT value.

In the next section we present different ways to model the environment for tighter timing analysis.

III. Modelling the Environment

The complexity of model checking without environment modelling is linear in the size $M$ of the timed automaton (TA) to be analysed [11]. With the addition of the environment information, the size of the TA to be analysed increases. Typically, whenever a data input that can have n different values is added to a TA of size $M$, the combined size of the composed TA grows to $M \times n$ in the worst case. We now show that using different representations for environment inputs (clocks or variables) can have different effects on the size of the composed TA, and consequently affect the time taken for static timing analysis differently. We have also implemented these different representation using the UPPAAL model checker, and the modelled environment runs parallel to the TA of the program, ensuring precise timing analysis.
A. Using Variables

Variables are typically modelled as bounded integers and their values are updated and used during verification. For example, the environment input sensor2 of Fig. 1, the data lies between (70..94), there are 25 possible values. Fig. 3 shows how this range is captured by the environment model. First, the variable sensor2 is initialised with a value of 70, its smallest valid value. The upper bound of 94 is ensured by the conditional statement sensor2 < 94. Once this condition is violated, the variable cannot be incremented further. The non-determinism between the two exit transitions ensures all the values between the interval 70..94 are computed by this environment model, thus precisely capturing the exact bounds on the input.

Every additional data variable with a range of \( n \) valid valuations, in the worst case, increases the size \( |M| \) of the SUT by multiplying it by \( n \). This also explains why unbounded variables are typically disallowed by formal tools. At the same time, the availability of precise variable valuations can help prune infeasible paths during timing analysis. In some cases, precise values of some variables may not be required, and clocks, as described below, may be a better representation for such variables.

B. Using Clocks

Real-valued clocks enable abstracting the values of a variable into regions, using region graphs \([2]\). A state or transition involving a clock is paired with all possible regions obtained from the bounds of the clock. The number of different values for a clock are essentially the number of regions obtained from its bounds. For example, the environment input reset of Fig. 1, whose value lies between (0..20), there are 21 possible values. Lets assume the clock reset has 3 regions: reset<2, 2<=reset<=18, and reset>18. A comparison reset<=10 on line 38 divides the region 2<=reset<=18 further into 2 regions 2<=reset<=10, and 11<=reset<=18.

Figure 4. modelling the Environment input reset as a clock.

In contrast, if reset was modelled as a variable (as shown in Fig. 3) instead of a clock, SUT would explore all the 21 possible values (0 to 20) of the input. Using clocks it would only explore two values, one greater than 10, and another less than 10. Hence, exploring both path of the conditional statement on line 38. Thus, it can be seen why clocks may help us achieve a speed-up in the calculation of WCRT.

However, clocks have limitations and are not as expressive as variables. The nature of clocks means that we cannot carry out precise verification in certain cases, and hence must use variables. For example, line 12 of Fig. 1 executes an average function where the input sensor1 is treated as an integer. Since clocks capture time and not equivalent to a variable, we cannot perform operations such as multiplication and division. Hence, in this model where we use only clocks, inputs sensor1 and input sensor2 that cannot be modelled as clocks are abstracted away. This leads to faster analysis time at the cost of tightness of the WCRT estimate.

C. Using Clocks and Variables

The higher expressiveness of variables and better abstraction of the clocks, leads us to propose a new intermediate approach. First we classify all environment inputs into two categories. If all conditional statements over an input can be evaluated even if that input is modelled as a clock, we choose to use a clock to model that input. More precisely, only if the left hand side (LHS) of a conditional statement is an environment input, and the RHS of the conditional statement is a constant, we classify such input as map2clock. However, if the RHS is a variable in some cases, and/or a constant in other cases, we classify such inputs as map2Variables. For the running example, since reset was only evaluated on line 38 against a constant of 10, clocks are used to model this input. The rest (sensor1 and sensor2) are mapped as variables. This new intermediate approach is shown in Fig. 5. This makes the timing analysis of this model faster than the variable model, but provides the same WCRT value.

D. Qualitative Comparison

Tab. II contains a comparison of the three techniques described above. It can be seen that while variables are more expressive than clocks, they slow down the timing analysis by increasing the size of the TA being analyzed. Using a combination of clocks and variables (based on the heuristic described above), provides the same high expressiveness as variables, and at the same time reduces the time taken for static timing analysis by reducing the size of the TA (as compared to using only variables).
### Table II

**QUALITATIVE COMPARISON OF THE PRESENTED APPROACHES FOR MODELLING ENVIRONMENT.**

<table>
<thead>
<tr>
<th>Factor</th>
<th>Variables</th>
<th>Clocks</th>
<th>Clocks-Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressiveness</td>
<td>high</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>Tightness of WCRT</td>
<td>tight</td>
<td>lax</td>
<td>tight</td>
</tr>
<tr>
<td>Analysis Time</td>
<td>highest</td>
<td>lowest</td>
<td>medium</td>
</tr>
</tbody>
</table>

### IV. Benchmarking

In this section we present a set of experimental results to show the applicability of the heuristic proposed in the previous section. We compare the effectiveness of the proposed approach with two techniques of using only clocks and only variables, presented in section III. We also present the scalability of the presented technique by comparing tightness, number of states explored, and analysis time. For these experiments, we have selected the benchmarks presented in [3], [4].

All programs execute on the MicroBlaze processor [15] along with its compilation tool chain as the target platform. For a given PRET-C benchmark, we generate the assembly files with the mb-gcc compiler, and then extracted the TCCFGs (as described in section II-B).

For each benchmark, from its TCCFG, we generate four different UPPAAL models. First model does not take the effect of environment into account, and is called called none. The second model, called clocks models all inputs as clocks (section III-B). The third model, called variables models the environment as described in section III-A. Finally the last model, called clock-variables models the environment under the heuristic presented in section III-C.

Table III presents the comparison between these four models. The first column of Table III lists the set of PRET-C benchmarks. The second column presents the estimated WCRT based on the models, called none. The third column presents the number of states explored during the analysis on the none models and time taken to analyse the WCRT is presented in column four. Similarly, rest of the table presents results for estimated WCRT, number of states explored and time taken to analyse (in seconds).

We compute the WCRT of an UPPAAL model, by model checking a property of the form \( AG(gtick \Rightarrow WCRT \leq val) \) (where \( val \) is an estimate of the WCRT), using a simple binary search method described in [11]. For computing the number of states explored and to measure the analysis time, we ran the UPPAAL command called memtime with the aggressive state-space reduction option.

Fig. 6 presents the WCRT estimates for the four models. On an average, when compared to models none, the WCRT estimate of a program improves by 18%, when clock based technique is used to model the environment. The WCRT estimate is further improved to 34%, when either variables based or clock-variables based technique is used. From Fig. 6, we can observe across all benchmarks, both the models variables and clock-variables have the same WCRT estimates. This is because both model the environment with the same level of abstraction (as described in section III-D). To investigate which is a better technique, we examine the number of states explored, and the time taken to analyse.

Fig. 7 presents the number of states explored for each benchmark, by both variables and clock-variables models. On an average, the clock-variables explores 91% less states than variables. This significant difference is due to
the fact that, UPPAAL performs symbolic abstraction over clocks, reducing the number of states explored. In comparison, for variables, UPPAAL performs exhaustive exploration, resulting in a much larger number of states explored. This difference in states explored has a direct effect on the time taken for WCRT analyse. Since the number of states explored has direct correlation to the time taken to explore path. The time taken to analyse the using clock-variables is significantly less than the variables technique. Fig. 8 presents the time taken to analyse, and on average the time taken clock-variables is 90% faster than variables. This gain in analysis time is directly proportional to the gain in the number of states explored.

V. Conclusions

In this paper, we presented three approaches for modelling the environment of a real-time program to obtain precise static timing analysis. We proposed a heuristic that models each environment input as either a variable or a clock, based on the control flow analysis of the program. Experimental results show that this heuristic provides the best balance between state space abstraction and precision of timing analysis.

Future directions for this work include exploring symbolic representations of the environment model to further reduce analysis time. We also endeavour to explore more detailed environment models to achieve tighter static timing analysis.

References


