Model Generation For Temporal Properties Of Reactive Components

Andrew Moss
Computer Science Department
University Of Bristol
moss@cs.bris.ac.uk

Henk Muller
Computer Science Department
University Of Bristol
henkm@cs.bris.ac.uk

ABSTRACT

We present a new static analysis that generates a model of the temporal behaviour of a reactive component. The component is specified in machine code - this makes the analysis applicable to the legacy code and the output of any compiler. Our analysis is an abstract interpretation [2] of the program that computes the possible periodicities of instructions within a nonterminating program. The machine code is transformed into an abstract instruction set that retains only the timing characteristics of each instruction. We show how a simple abstraction of the timing state allows an abstract interpretation to construct a finite model of the components temporal behaviour. This model is useful for manual verification of a simple component or be used to construct an automatic verification through model-checking. The abstraction of time uses a relative measure of time; time is measured since the flow of control passed certain nominated positions.

1. INTRODUCTION

We describe the class of systems applicable to our analysis by using Pnueli's [10] categorisation of systems into two types; reactive systems that are non-terminating and must hold an ongoing dialogue with their environment, and transformational programs that take an initial input, execute for a (hopefully) finite amount of time and produce an output. Within this dichotomy we are concerned with reactive systems. We further divide this class into systems that are time-triggered — that interact with their environment at specific points in time — and those that are event-triggered — that interact with their environment at non-specific times but with specific types of interaction. Our analysis is applicable to time-triggered reactive systems.

The motivation for our analysis is a simple domain of reactive systems; sensor components from a wearable computer [13]. These are composed of a microcontroller and an interface to a hardware sensor that samples the environment at precise moments in time. This combination of a programmable processor executing a well defined instruction set and an external interface with exactly defined timing characteristics is quite common in real time systems and so our analysis has wide applicability. Our concern is the verification of the timing behaviour of the components, not their logical functionality. In particular, we will note that that in distributed systems where an embedded component has to communicate with other embedded components at precise times, we may consider the other components to be simpler sensors and actuators without a loss of generality. By considering interactions with other components to be of the same nature as interactions with sensors we can analyse any system composed of processor and sensor components within our domain of time-triggered reactive systems.

The systems that we analyse are repetitive, but not strictly periodic. Each is constructed from repetitions of a terminating procedure, but the procedure may not execute in a uniform amount of time, leading to a varying period for the system. Each statement within the program has a set of periods between successive executions. These periods are important because certain statements within the procedure cause side-effects that are externally visible. These statements are modifications and the reading of bits within control registers in the microcontroller. Setting bits in these registers drives pins on the microcontroller that control external hardware. Reading these bits polls the current state of external pins. By analysing the set of periods that each statement has, we are analysing the set of periods of these visible interactions. The set of all sets of statement periods forms a model of the programs timing behaviour that can be used to verify its temporal properties.

We are generating a model of the times at which each instruction in the program can be executed.This model consists of a set of timing configurations. Each configuration is a location within the program and a clock counter measuring discrete time since the program started execution. This model allows verification that the externally observable events within the program can only occur within a valid set of times.

For the program under analysis, we model the flow of control as a sequence of non-deterministic choices. In the instruction set that we are analysing all of these choices are binary. We will assume that either branch can be taken by the program. If we make this assumption for all branches in the program

then the number of timing states is infinite. This follows from the observation that the looping branch can always be chosen, and that as each loop takes a positive non-zero number of cycles new states are being generated for each iteration of the loop. The key transformation in our analysis is to partition the set of branches into those that form loops and those that do not. For looping branches we will only make a non-deterministic choice on the first execution. After they have been executed once we will deterministically choose the non-looping branch. As the program is of finite length, and each part can only be repeated a finite number of times, this ensures that the space of timing-states that we compute is finite.

It is only necessary to consider the first execution of a looping branch as a non-deterministic choice. This is because we are constructing the set of timing traces for all possible executions of that iteration. This set of traces therefore contains the possible executions of all iterations of the loop.

We have taken the problem of constructing the space of possible timing states and partitioned it into several smaller problems. We have the problem of deciding which branches in the control flow graph form cycles; this is decidable. We then construct a finite representation of the timing paths through this graph without executing each cycle more than once. In order to use these paths to construct the set of timing-states we would have to decide when each loop is taken, which is an undecidable problem. If we could decide analytically whether or not looping conditions were taken then we could decide if the program halted. This restriction means that our analysis is limited to programs only containing loop expressions that we can analyse. There is a wealth of material [5, 4, 15, 1] in the literature devoted to analysing the bounds of different type of loop expression, we shall not repeat this material here but simply assume that the programs we are interested in only contain loop expressions amenable to existing analyses.

This leaves the last problem of deciding which branches are taken within decisions. We contend that in order for the program to be correct, if there is a decision to be taken, then the timing behaviour of both branches must be correct otherwise the program can fail. If only one branch is meant to be taken then it should be a piece of straight-line code and not a logical decision. So the over-approximation that we make is one that a correct program should satisfy within the context of embedded systems.

The program under consideration may decide its actions on the basis of the information that is communicated from the sensors, but by considering the entire set of possible timings between observable events we may safely discard the process which selects one of these timings through some form of logical decision on the data contained within these communications. This set of possible timings is not as precise as one that also analyses the logic that the device performs, but it is a conservative over-approximation of the set. For verification purposes we wish to ensure that it is not possible for the device to operate outside of some specification of the times between observable events, thus the logical behaviour of the device is irrelevant for our purposes as the over-approximation of its behaviour must lie within the bounds defined by the specification.

2. RELATED RESEARCH

The problem of constructing a model of the execution times of a program's statements is generally undecidable. In order to form a decidable problem it is necessary to make assumptions about the type of program being analysed, and to place appropriate restrictions on what the program can do within the context of these assumptions. The set of assumptions that are made will define not just the restrictions on what the program can do, but also the form that the result of analysis will take.

If we assume that the program we are analysing is transformational, rather than reactive, then we will assume that correct programs terminate in order to produce a result. In this case we are less interested in the structure of the timing model, and more interested in analysing whether the extreme bounds of the model fit within a set of constraints. The more common case is checking that the upper-bound of execution time is within a constraint. This case is known as WCET (Worst Case Execution Time).

2.1 Major results in WCET

The initial results [6, 14, 8, 9, 11] in the field of WCET, follow this assumption of termination in order to guarantee a finite length worst case path. The restrictions placed on programs under analysis are that loop bounds are known a priori, there are no dynamic data-structures, unbounded recursion or dynamic references to functions.

WCET analysis is now a mature field [12] with the above model of program analysis refined to consider many architectural features such as RISC, pipelines and caches. Loop bounds can be automatically annotated in a wide range of cases. The precision of results has been increased through the elimination of infeasible code paths and a tighter mapping from source annotations to the underlying assembly language.

2.2 Differences from WCET

Our method offers an improvement over WCET analysis in several ways; WCET produces an upper bound on the execution of a piece of code - we are interesting in the exact set of possible times between events rather than an upper-bound upon them. This allows verification of the correct variances in times between events. This exact set is required in some scheduling problems that are encountered in embedded systems [3]. As WCET analysis considers the total execution time of a software component it is not capable of providing event to event timings across loop iteration boundaries within the same component. We are interested in the periodicities of non-terminating systems where-as WCET assumes that systems are terminating.

Our results are strictly more conservative than WCET in the context of how much of the potential state-space they consider to be reachable. They provide a more finely structured result in the context of how they represent that state-space which provides the answers to more precise questions such as event to event timings.

2.3 Applicability of WCET results to model generation

WCET analysis is split into three phases that perform different transformation upon the source program. The names of the three phases can vary between researchers although the purpose of each phase is generally the same:

Low-Level Analysis extracts the cycle accurate timing information from the low-level code representation, usually either machine code or an intermediate form within a compiler.

Flow Analysis is concerned with defining flow-facts for the program. These are infeasible paths which exist in the code but cannot be taken because of logical effects, variable bounds and hence loop bounds.

Calculation combines these two sets of information about the program into an estimate of the worst case execution time.

The analysis that we describe in this paper is analogous to a low-level analysis, but based on a different set of assumption than those used in WCET. We are concerned with the extraction of cycle accurate path times from a program. Within the context of embedded systems these are sufficient to answer some useful questions about the system's simple temporal properties. In order to answer more complex questions about the temporal properties of reactive components we need to implement a similar flow analysis, and then construct equations to describe the periodic behaviour of the component. The extraction of flow facts can use existing source-level techniques from the field of WCET. The final generation of equations to describe the periodic behaviour is substantially different. The description of these two phases, as they apply to reactive components, is beyond the scope of this paper and will be the subject of future work.

3. EXAMPLE CODE

The microcontroller that we use as an example in this paper is the PIC-16F84 [7]. This microcontroller is of interest because it offers a good ratio of MIPS to power usage, which makes it a common choice of controller within the wearable and robotics communities. The instruction set of the PIC gives a low code density as all operations have to be performed on a single working register rather than in any register as with RISC instruction sets. This simplicity gives the PIC ultra-low power consumption, using just 2 mW of power when active.

The code that we use as an example for our analysis performs the transmission of a byte on a serial interface. This code is shown in Figure 1. The serial interface is a simple teletype protocol with a start bit, 8 bits of data, and a stop bit. This simple protocol is useful for a system made up of distributed components as it doesn't require a synchronised clock at both end points. Instead, the start bit is used to synchronise the sender and receiver and then as long as the drift between the transmission and receiver clocks is within a small tolerance(5% of the transmission time) the data will be communicated correctly.

```
XMIT
    MOVWF  SER_TX
    MOVLW  NUMBIT+1
    MOVWF  BITCNT       ; Preset data bit counter

;   Send the start bit

    BCF    PORTB,TXD    ; Set start bit level
    GOTO   XMITC        ; Wait for start element
                        ; to go
;   Set the transmit data level from the carry and
;   wait for an element

XMITA RRF    SER_TX,1   ; Clock shift register RIGHT
                        ; through carry
    BTFSC  3,0          ; If data (carry) is '0',
                        ; skip
    GOTO   XMITB
    BCF    PORTB,TXD    ; Data is '0'
    GOTO   XMITC
XMITB
    BSF    PORTB,TXD    ; Data is '1'
XMITC
    ; call WAITEM       ; Wait for the element to go
    NOP
    ;NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP

;   Count the elements as they are sent

    MOVF   BITCNT,1     ; Zero if just sent the stop
                        ; bit
    BTFSC  3,2          ; Skip next if bit count is
                        ; not zero
    RETURN              ; Exit from XMIT

    DECFSZ BITCNT,1     ; Dec. bit count, skip if zero
    GOTO   XMITA        ; Loop until all bits are sent

;   Bit count has zeroed, send the stop bit

    BSF PORTB,TXD       ; Set stop bit
    GOTO   XMITC        ; Wait for the stop bit to go
```

Figure 1: Sample serial transmission code

The sample code relies on fine-grained timing synchronisation to operate the serial line at the correct rate. The sample shown is for a serial line operating at 115200 baud. The microcontroller uses a 10MHz crystal to operate at 2.5MIPS. This requires the line state to be changed every 21.7 clock cycles, the closest we can manage on a discrete clock is every 22 cycles which is within the error tolerance.

To achieve this cycle accurate timing the sample code uses NOP instructions to pad the length of time required to iterate the main loop. The main loop is XMITC which executes 10 times, once for each bit. The initialisation code jumps into the middle of this loop, rather than entering through the head in order to produce the right period between the instructions that control the hardware interface. The commented out sections are evidence of the trial and error required in order to produce the correct temporal execution.

4. ABSTRACT INSTRUCTION SET

Each instruction within an ISA modifies the state of the processor. These changes in state can be broadly divided into three categories, and each instruction will then have effects in each of these categories:

Functional effects, such as modifying the contents of registers or flags.

Temporal effects which are the difference in the processors cycle counter from the beginning to the end of the instruction.

Control flow effects which modify the program counter (pc).

For the purposes of this analysis we are interested solely in the temporal and control effects that each instruction causes. By ignoring the functional effects of the instruction set we reduce the complexity of the analysis significantly, both in the technical sense that we reduce the size of the fixed-point that we compute, and also in the informal sense that it makes our analysis easier to understand and implement.

The PIC ISA contains 35 instructions, which provide data movement, bit manipulation, logical testing and control-flow manipulation. The instruction set does not contain the usual predicated jump instructions. The only jumps used are unconditional, instead there are bit-testing instructions which will test the status of a bit in a register and conditionally skip the next instruction if it is set/unset. These skip instructions can be combined with jumps to construct the normal predicated branches by testing the processor flags which are contained within a control register.

When we ignore the functional effects of the ISA then these 35 instructions form 5 classes of equivalent control and temporal effects. Our analysis operates directly upon these equivalence classes which we term the abstract instruction set. For the PIC these equivalence classes are:

SINGLE Takes 1 cycle and pass control to the next address (pc+1)

$$(pc, t) \Rightarrow (pc + 1, t + 1) \quad \text{when } P(pc) = \textsc{Single}$$

$$(pc, t) \Rightarrow (-1, t + 2) \quad \text{when } P(pc) = \textsc{Return}$$

$$(pc, t) \Rightarrow (pc + 1, t + 1), (pc + 2, t + 2) \quad \text{when } P(pc) = \textsc{Skip}$$

$$(pc, t) \Rightarrow (tar, t + 2) \quad \text{when } P(pc) = \textsc{Jump}(\text{tar})$$

$$(pc, t) \Rightarrow (-1, t + 2) \quad \text{when } P(pc) = \textsc{Call}(\text{tar})$$

Figure 2: Abstract Instruction Semantics

RETURN Takes 2 cycles and return to the top address on the stack

SKIP Takes either 1 or two cycles and pass control to either pc+1 or pc+2

JUMP(n) Takes 2 cycles and pass control to the address n encoded within the instruction

CALL (n) Takes 2 cycles, passes control to address n and pushes the pc onto the stack

The operational semantics of these abstract instructions are shown in Figure 2. These transitions are between configurations composed of (a, b) where a is the program location and b is the number of clock-cycles since the program started execution.

In order to measure the length of program traces we must firstly construct a model of the timing characteristics of the ISA. In the case of the PIC micro-controller this model can be derived entirely from local effects within the code for the majority of the instruction set. Abstracting the functional characteristics of the instruction set we are left with the length of time to execute an instruction, and the locations that control is transferred to by the execution of that instruction. We do not consider the selection of which transition occurs in a given state which folds the instruction set into a simpler non-deterministic model. Each instruction in the abstract instruction set is a representation of a set of concrete instructions. Each transition is labelled with an integer number of clock-cycles, but there are no guards on the transitions.

The Call and Return instructions form transitions that are dependent upon a state (in this case the stack of return addresses). We do not represent this state within the timing states of the program. Instead we consider each locally reachable region of the program graph (subroutine) within an independent analysis and then use a call-graph to propagate information between them. We only cover the computation of a single reachable region within this paper. The reason for this is that to completely characterise the timing properties of a reachable region and then substitute it back into the calling region we need to use a flow analysis

to deduce loop bounds and then generate equations that describe the complete timing behaviour of the region. This is beyond the scope of this paper. The consequence of this choice is that the analysis cannot determine the number of clock cycles that calls to subroutines will require and so is not suitable for code containing calls.

Conversion from PIC assembly code into the abstract instruction set is a simple many-to-one mapping for each location. Locations and therefore code-lengths are preserved when the program is represented in the abstract instruction format. The mapping of the sample program into the abstract instruction set is shown in Figure 4.

5. TIMING CONFIGURATIONS

The configurations within an abstract interpretation are compositions of the state and the location that the state is valid at. In our analysis these states are the temporal state. The concrete temporal state is an integer, representing the number of clock cycles since program instantiation. The set of configurations with a common location then give the times that a statement can be executed at. This is potentially infinite for a repeating component.

We abstract these temporal states by using a relative clock. Instead of the number of clock cycles since the program started execution we use the number of clock cycles since the program passed a recording point. This gives a state that is a tuple of two integers; (a, b) where a is the number of clock cycles and b is the location of the recording point.

If we chose our recording points arbitrarily then we will not reduce the cardinality of the state space within the analysis. If there is a loop between the recording point and the current point of execution then arbitrarily large time values can be recorded by continuing to chose the branch that iterates the loop. Our solution is to make the recording points the backward edges of loops within the program control flow graph. This ensures that *every* loop that could be taken between the recording point and the current point causes a new recording point to be set. Therefore it is no longer possible to construct arbitrarily large lengths of time in the state-space and the size of the state-space must be finite. This follows from the observation that there are only finitely many locations within the program, and the distance from a recording point to another location without iterating a loop is bounded by the length of the program. Intuitively, our abstraction folds all iterations of a loop onto the first iteration.

This is not a strict abstraction. We are folding the states associated with all iterations of every loop, in the program, onto the first iteration of the relevant loop. However we are also recording more information about the path taken to reach the current location than exists in the concrete state, as each state contains the last loop back-edge traversed. This means that states that were equal in the concrete domain are now differentiated in the abstract domain. This increases the size of the fixed-point generated slightly but the extra information recorded is useful for the determination of flow-facts in a later separate analysis.

If we represent our program by a pair of functions P, and

$$
\begin{array}{lcl}
(pc, t, l) & \Rightarrow & (pc+1, t+1, l) \\
 & & \text{when } P(pc) = \textsc{Single} \\
(pc, t, l) & \Rightarrow & (-1, t+2, l) \\
 & & \text{when } P(pc) = \textsc{Return} \\
(pc, t, l) & \Rightarrow & (pc+1, t+1, l), (pc+2, t+2, l) \\
 & & \text{when } P(pc) = \textsc{Skip} \\
(pc, t, l) & \Rightarrow & (tar, 2, pc) \\
 & & \text{when } P(pc) = \textsc{Jump}(\text{tar}) \wedge \text{L(pc)} \\
(pc, t, l) & \Rightarrow & (tar, t+2, l) \\
 & & \text{when } P(pc) = \textsc{Jump}(\text{tar}) \wedge \neg\, \text{L(pc)} \\
(pc, t, l) & \Rightarrow & (-1, t+2, l) \\
 & & \text{when } P(pc) = \textsc{Call}(\text{tar})
\end{array}
$$

Figure 3: Abstract Instruction Semantics On Abstract Time

L, such that P maps locations to an element of the set of abstract instructions, and L decides whether a location is a loop back-edge then we can define a parameterised transition system that gives the semantics of the abstract instructions when applied to abstract configurations. This is shown in Figure 3.

The Skip instruction class is a set of instructions that tests logical decisions and passes control along one of two transitions. It implements a guarded choice of next instruction in the program that is used to build more complex control-flow structures such as guarded jumps. The non-determinism in our transition system is contained in the transition from the location of a Skip instruction to both the following instruction, and the instruction after it.

Given this definition of a transition system parameterised by a program function P and a program function, we can generate the configuration transition system for the timing of the program. We demonstrate this in Figure 4 which shows the P function for our sample program and the application of the parameterised transition system to this instance of P. This combination produces the transition system shown.

6. COMPUTING REACHABILITY DISTANCE

The transition system that we construct for a program consists of tuples of the form (f, t, l). These give transitions in the program control flow graph and a distance in clock cycles of making these transitions. Given the set of locations that are loop-edges we can now compute the set of temporal configurations for the program.

For each configuration of (a, c, a') we apply the relevant transitions from the transition system to produce a set of new configurations. These are the transitions where $f = a$. When the location that control is being transferred from, f, is a loop back-edge we reset the recording point a' to a and reset the time to 0. Each new configuration is therefore $(t, c + l, a')$ when not reseting the recording point, and (t, l, a) when reseting the recording point.

P(Sample Program)		Transitions applied to P(Sample Program)
n	$F(n)$	$(from, to, len)$
0	Single	(0,1,1)
1	Single	(1,2,1)
2	Single	(2,3,1)
3	Single	(3,4,1)
4	Jump(11)	(4,11,2)
5	Single	(5,6,1)
6	Skip	(6,7,1),(6,8,2)
7	Jump(10)	(7,10,2)
8	Single	(8,9,1)
9	Jump(11)	(9,11,2)
10	Single	(10,11,1)
11	Single	(11,12,1)
12	Single	(12,13,1)
13	Single	(13,14,1)
14	Single	(14,15,1)
15	Single	(15,16,1)
16	Single	(16,17,1)
17	Single	(17,18,1)
18	Single	(18,19,1)
19	Single	(19,20,1)
20	Single	(20,21,1)
21	Single	(21,22,1)
22	Skip	(22,23,1),(22,24,2)
23	Return	(23,-1,0)
24	Skip	(24,25,1),(24,26,2)
25	Jump(5)	(25,5,2)
26	Single	(26,27,1)
27	Jump(11)	(27,11,2)

Figure 4: Creation of Temporal Transition System

((-1),14,9)	((-1),14,27)	((-1),18,0)	((-1),19,25)
(0,0,0)	(1,1,0)	(2,2,0)	(3,3,0)
(4,4,0)	(5,2,25)	(6,3,25)	(7,4,25)
(8,5,25)	(9,6,25)	**(10,6,25)**	(11,2,9)
(11,2,27)	(11,6,0)	(11,7,25)	(12,3,9)
(12,3,27)	(12,7,0)	(12,8,25)	(13,4,9)
(13,4,27)	(13,8,0)	(13,9,25)	(14,5,9)
(14,5,27)	(14,9,0)	(14,10,25)	(15,6,9)
(15,6,27)	(15,10,0)	(15,11,25)	(16,7,9)
(16,7,27)	(16,11,0)	(16,12,25)	(17,8,9)
(17,8,27)	(17,12,0)	(17,13,25)	(18,9,9)
(18,9,27)	(18,13,0)	(18,14,25)	(19,10,9)
(19,10,27)	(19,14,0)	(19,15,25)	(20,11,9)
(20,11,27)	(20,15,0)	(20,16,25)	(21,12,9)
(21,12,27)	(21,16,0)	(21,17,25)	(22,13,9)
(22,13,27)	(22,17,0)	(22,18,25)	(23,14,9)
(23,14,27)	(23,18,0)	(23,19,25)	(24,15,9)
(24,15,27)	(24,19,0)	(24,20,25)	(25,16,9)
(25,16,27)	(25,20,0)	**(25,21,25)**	(26,17,9)
(26,17,27)	(26,21,0)	(26,22,25)	(27,18,9)
(27,18,27)	(27,22,0)	(27,23,25)	

Figure 5: Fixed-point for sample program

The union is calculated of the current set of configurations and the set of unique new configurations, created by applying the transitions to the current set. This union is computed until a fixed-point is reached. The existence of this fixed-point is guaranteed by the finite number of locations and possible states.

This fixed-point contains the set of all time distances from recording points (the program start point and the set of loop back-edges) to program locations. This result can be seen as a reachability distance, showing not only which locations are reachable, but also the distance in time to reach that location.

7. RESULTS

Applying the analysis to our sample program gives several useful timings which we can use to verify the correctness of its execution. These are the period of the main loop executed for each bit, and the phase of the bit setting operations within this loop.

The fixed-point of configurations for the sample program is shown in Figure 5. We have highlighted the relevant configurations within this set to illustrate the correctness of the program with respect to the timing criteria that we now set out. The actions that we wish to verify the temporal properties of, are carried out by the target instructions within the program. The configurations that are relevant show the time distances between the back-edge of the transmission loop back-edge and the target instructions.

There are three temporal properties of the bit transmission loop that we verified. These properties all concern the main transmission loop, we wish to ensure that it has a uniform period, that the period is correct, and that the phase of the observable actions within this loop are correct and constant.

7.1 Uniform Loop Period

The main transmission loop (with back-edge at location 25) should have the same period upon each iteration. The set of configurations that form the fixpoint of the timing behaviour for the program contains the periods of each loop within the program. These periods are encoded as timings from the back-edge of a loop to the same back-edge node. We can extract this set through a filtering operation upon the fixpoint set. This filtering operation maps one set of configurations onto a second set of configurations according to a boolean operator on configurations. Filtering sets in this way using a higher-order function is a common idiom within functional languages and so we present the pseudo code for this operation in the syntax of Haskell. If this set of loop periods is a singleton set then we have verified that the loop has a uniform period across all executions of the program.

```
loopTimes :: Set Configurations -> Int ->
             Set Configurations
loopTimes fp loop = filterSet cond fp
   where cond (l,f,t) = (l==loop && f==loop)

uniform :: Int -> Set Configurations -> Boolean
uniform loop fp = 1==setSize (loopTimes loop fp)
```

Upon the fixpoint of the example program, the filtering expression (`filterSet cond fp`) yields the result $\{(25, 21, 25)\}$ which is a singleton set. Hence the expression for the example program (`uniform 25 progfp`) is true proving that the main transmission loop has a uniform period.

7.2 Correct Loop Period

It is a common requirement of time-triggered reactive programs that we must verify the period of their loops which contain interactions with the environment. In order to perform this verification we perform the same filtering of the fixpoint as the previous section, and then map the configurations into the set of integer periods.

```
periods :: Set Configurations -> Int -> Set Int
periods fp loop = mapSet extr (loopTimes loop fp)
    where extr (l,f,t) = t
```

Performing this operation (`periods 25 fp`) on the example program yields the set $\{21\}$ which gives the period in processor cycles of the main transmission loop. This deviates from the target of 21.7 cycles and executing the program over 10 bits will take 3 cycles too few. This is within the specified tolerance and so the length of the loop period has been proven correct.

7.3 Constant Operation Phase

The bit manipulation instructions at locations 8 and 10 in the example program form an operation that interacts with the environment. These two instructions are on mutually exclusive program branches, so that exactly one of the two will execute on each iteration of the loop.

The distance in time from the execution of the back-edge of the loop to the visible operation is the phase of the operation within the loop. For some operations, such as the one shown in the example program, this phase should be constant in each iteration. In the example program a change in phase of the bus actuating operation would introduce a bias into the sampling of the line state.

In order to verify that an operation has constant phase we must ensure that the set of locations forming that operation all have constant distances from the loop back-edge in the program fixpoint. This is a similar filtering operation to the check for constant loop period. For a given set of instruction locations and a loop location we must compare the set of matching distances within the fixpoint.

```
phaseInst :: Set Configurations -> Int -> Int ->
             Set Int
phaseInst fp loop inst = mapSet mop
                                (filterSet cond fp)
    where cond (l,f,t) = (l==inst && f==loop)
          mop  (l,f,t) = t

phaseOp :: Set Configurations -> Int -> Int ->
           Set Int
phaseOp fp ins loop = unionSets (mapSet phaseInst
                                        ins)
```

```
constantPh :: Set Configurations -> Set Int ->
              Int -> Boolean
constantPh fp ins loop = 1 == setSize (phaseOp
                                       fp ins loop)
```

The expression `constantPh fp {8,10} 25` evaluates to false, proving that the phase of the instructions differs causing a bias in the width of the pulses that is dependent upon the data being transmitted.

7.4 Correct Operation Phase

The last type of verification that we shall show is checking the actual phase of operations within the loop. In the example that we have presented this would be necessary in order to determine how much bias is introduced into the width of the pulses being transmitted. In other programs it may be necessary to check the phase of separate operations in order to verify a relationship between them, e.g. when polling sensors that transmit data encoded into the width of pulses modulated to some duty cycle.

The code already presented is sufficient to determine the set of phases that an operation can occur at. The expression `phaseOp fp {8,10} 25` evaluates on the example program fixpoint to $\{5, 6\}$ showing that the phase of the operation differs by up to one cycle. This determines the amount of bias that this error in the program introduces.

8. CONCLUSION AND FUTURE WORK

We have described a new form of timing analysis that can be applied to the programs of non-terminating reactive components. This analysis calculates a fine-grained timing metric across the locations within the program. This information can then be used to verify the correctness of a program's temporal behaviour. We have successfully applied this analysis to a microcontroller, and used it upon a real-world example to verify the correctness of its timing behaviour.

For more complex programs the direct inspection of the fixed-point that we have used would not be feasible. For nested loop structures the timing distance measured in the analysis does not take into account the iterations of the inner loop. In future work we will address these short-comings by showing how to construct timing equations which generate the sets of timing distances for locations in more complex programs.

The analysis that we have presented can be seen as the first of three states in a full timing analyser. The second stage will detect structural properties of the program, such as loop bounds and infeasible paths through the program. The third stage will combine this structural information with the low-level timing distances to create generating expressions for the execution times of locations within the program.

The problem of establishing loop bounds is orthogonal to the problem that we have solved, of deriving timing distances across program code. However, both need to be combined in order to generate precise accurate expressions for the temporal behaviour of instructions within a program. We have not fully explained the integration of the call and return

instructions in this work as in order to propagate timing information through the program call-graph this complete description of timing behaviour is required. The problem of fully analysing calls and returns in source programs will be the focus of future work. This future work will integrate all three stages described above.

Acknowledgements

We would like to thank the anonymous reviewers for their comments which helped to improve the structure and presentation of this article.

This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 ASAP project.

9. REFERENCES

[1] Zahira Ammarguellat and W. L. Harrison, III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 283–295. ACM Press, 1990.

[2] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977. ACM Press.

[3] A. G. Dean and J. P. Shen. Techniques for software thread integration in real-time embedded systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, page 322. IEEE Computer Society, 1998.

[4] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proc. 4th Real-Time Technology and Applications Symp.*, pages 12–21, Denver, CO, June 1998.

[5] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):129–156, May 2000.

[6] E. Kligerman and A. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Trans. on Software Eng.*, 12(9):941–949, September 1986.

[7] Microchip. Pic16f84 data sheet. Technical Report DS35007B, Microchip Technology Inc, 2001.

[8] A Mok. Evaluating tight execution time bounds of programs by annotations. In *Proc. 6th IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–80. IEEE, May 1989.

[9] Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, 24(5):48–57, 1991.

[10] A Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. pages 510–584, 1986.

[11] P. Puschner and Ch. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[12] Peter Puschner and Alan Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[13] Cliff Randell and Henk Muller. Context awareness by analysing accelerometer data. Technical Report CSTR-00-009, Department of Computer Science, University of Bristol, August 2000.

[14] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Trans. on Software Eng.*, 15(7):875–889, January 1989.

[15] Robert A. van Engelen and Kyle A. Gallivan. Tight non-linear loop timing estimation. In *Proceedings of the 2002 International Workshop on Innovative Architectures*, pages 21–26, January 2002.