Worst Case Reaction Time (WCRT) analysis techniques for synchronous program

Matthias Kremer
University of Kaiserslautern, Embedded Systems Group
m_kremer13@cs.uni-kl.de

Abstract

In order to synchronize execution between multiple threads of a reactive synchronous program, execution is divided into discrete computation blocks, called ticks. Synchronous programs guarantee determinism (safety) and reactivity (liveness). To avoid timing faults, it is important to determine the worst case tick length (longest time of any tick), also known as the worst case reaction time (WCRT). If the ticks of a synchronous program are scheduled using a clock with a period greater than or equal to the WCRT, it is guaranteed that the real-time deadlines are met. As a lot of research has gone into the worst case execution time (WCET) analysis of procedural programs and WCRT analysis received less attention, there is only a handful of techniques available for determining the WCRT of synchronous programs. Hence, the existing solution for the WCRT analysis often lead to massive overestimation and therefore are not suitable for accurate timed reactive systems. In this paper, I summarize three different approaches with different accuracy and speed to calculate the WCRT value.

The first algorithm presented determines the WCRT of programs written in the synchronous language Esterel and executed on the reactive processor Kiel Esterel Processor (KEP), which has an instruction set architecture supporting direct execution of Esterel programs. Here the WCRT value is given in terms of the instruction cycles of the KEP.

The other two approaches use a synchronous C based language called PRET-C (Precision Timed-C), develop different graph structures and automata where WCRT analysis is performed on.

1 Introduction

Embedded reactive systems continuously receive inputs from the environment and react accordingly by generating the corresponding outputs. As an upper bound of the required execution time to create these outputs is crucial for these systems, it needs to be guaranteed that the worst case execution time does not exceed this crucial upper bound. Hence such systems are called real-time systems, as they must meet precise timing requirements.

Commonly reactive application consist of several threads which perform different tasks. To synchronize the execution of these threads and that the compiler does not assign the processor to one thread only, the overall execution of the program is divided into global ticks.

Every thread’s execution is also divided by local barriers, called EOT (End Of Tick in PRET-C) or pause (used in Esterel). The computation done between these barriers is called a local tick. A global tick elapses only when all participating threads reach their respective barrier, meaning that their local tick has finished. This concept of a global tick is used to ensure precise timing of execution of all threads.
Furthermore it is ensured that the next tick is started only when all threads have reached their barriers. The execution of a local tick consumes time, mostly measured by clock cycles of the processor. Even if the local threads may vary in execution time, the global tick must be of a fixed length so that no timing faults occur. The task of the Worst Case Reaction Time (WCRT) analysis is now defined as finding the longest global tick of the synchronous program.

The first approach to this topic was done in 2008 and is presented in section 3.1, where programs written in the synchronous language Esterel are investigated. The algorithm presented there basically just sums up the WCRT of the individual threads to calculate the overall WCRT. Usually this turns out to be a gross overestimation. To understand how this algorithm works, it is necessary to introduce several concepts, including the Esterel language itself, the processor it is executed on (KEP) and an intermediate data structure (CKAG), on which WCRT analysis is performed. The algorithm is rather complex to come up with a relatively “easy” result, but the complexity is necessary to do a correct analysis.

Another approach was presented in 2009 using model checking. Here programs written in the synchronous language PRET-C are translated to several data structures like the TCCFG, which is then compiled to produce the input for the model checker. Further it is queried with a value val to check if val is equal to or less than the WCRT of the system. This approach is presented in 3.2.

A third approach done in 2011 addresses the problem of the high time consumption of the model checking approach. The proposed approach shows how WCRT computation can be solved by on-the-fly reachability analysis of a synchronous program. It also investigates PRET-C programs and uses the same data structures form the model checking approach. This is the last approach, which is presented in 3.3.

2 Related Work

Until the first paper [1] was released in 2008, all existing work in this field was concerning Worst Case Execution Time (WCET) analysis, which is about determining the execution time of sequential programs. Task periods and deadlines should be determined. As reactive synchronous programs execute in ticks, new algorithms needed to be developed to facilitate the need of meeting precise timing and therefore to calculate the maximum time one tick needs to execute. The very first approach was done by:


As this approach is the most easy one in terms of complexity of the solution, the calculated WCRT value is usually far too high to be relevant. To accommodate this, the model checking approach was presented in 2009. At least three other approaches ([10], [11], [12]) had been presented since these two, but as their complexity is exponential or NP hard, they are not discussed here.

[4] P. S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen. Tight WCRT analysis for synchronous C programs. In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES09), Grenoble, France, October 2009. IEEE. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. CASES09, October 1116, 2009, Grenoble, France. Copyright 2009 ACM 978-1-60558-626-7/09/10 ...$10.00. [4]
The third approach computes WCRT with a reachability analysis of the thread states and has some advantages over the model checking one. It does not produce more accurate results, but it has lesser complexity than model checking and therefore computes the WCRT value faster.


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. DAC 2011, June 5-10, 2011, San Diego, California, USA. Copyright 2011 ACM 978-1-4503-0636-2/11/06 ...$10.00. [9]

3 The Solution

All three algorithms are explained in this section including an example. The first approach ([1]) from 2008 can be found in 3.1, the second ([4]) from 2009 in 3.2 and the third ([9]) from 2011 in 3.3.

3.1 Max-Plus Approach

The basic idea of this approach is to calculate the maximum WCRT of each thread and then sum them all up to gain the overall WCRT for the whole program. In the following, two examples of programs written in the synchronous language Esterel are presented, the language itself together with the executing processor gets explained and how WCRT computation is preformed. The general algorithm can be found in the end of this section.

3.1.1 Esterel, KEP and CKAG

Esterel In order to carry out a WCRT analysis of reactive synchronous programs, one has to choose a programming language that provides direct and predictable support for reactive control flow patterns. In addition to that it also has to support concurrency and multiple forms of preemption (abort constructs) and scheduling ([1], page 66). One suitable candidate for this is the synchronous language Esterel, whose execution is by construction divided into logical ticks, which serves the overall purpose of the WCRT analysis. Communication within or across threads occur via signals. Esterel statements can either be transient or delayed. Transient statements do not consume logical time (e. g. emit or loop statement). If a statement is delayed, the execution is finished for the current tick. The most important delayed Esterel statement is pause, which is the barrier statement in Esterel. If execution reaches pause, the tick finishes and from there on the next one starts. Esterel forbids programs to perform a potentially unbounded number of statements within one tick. This results in the rule that there cannot be any instantaneous loops. Every loop body must contain at least on pause statement in order to prevent an execution of infinite statements in one tick ([1], page 68).

Esterel offers two types of abortion constructs. As soon as an abortion is triggered, the body of the abortion gets killed. A strong abortion kills its body immediately at the beggining of a tick, a weak abortion lets its body receive control for one last time and kills the body at the end of the tick.

Another important concept is Esterel’s parallel operator, ||, which spawns concurrent threads who execute
Figure 1: A simple sequential Esterel Example. The body of the KEP is annotated with line numbers L1-L6.

Figure 2: A concurrent Esterel example.
statements independently. Only if both spawned threads finished their execution, the main thread is freed out of suspension and continues with execution.

A simple example ExSeq of a sequential Esterel program can be found in Figure 1(a). It consists of a loop within a weak abort. Within the loop there is a pause statement (end of tick) and an emit statement, which in this case releases the signal R. This signal gets emitted continuously from the second tick on, until the weak abort is triggered. This happens when the signal I occurs, which lets the body receive control for one last time and so R is also emitted for the last time. Because the weak abort was triggered, the pause statement does not mark the end of a tick any more. Hence, within the same tick, the signal S is emitted and the program terminates. This behaviour can also be observed in the trace right below the program, where the input I occurs in the third tick. A more relevant example for the WCRT analysis is ExPar and can be found in figure 2(a), which loops continuously over two parallel threads. This program emits the signals R and S in the first tick. In the second tick, signal T is emitted and as both threads finished their execution, both get instantaneously restarted again and signal R and S are emitted, too.

**Kiel Esterel Processor (KEP)** Besides choosing a suitable programming language, finding the right execution platform is crucial for deterministic results of the worst case reaction time analysis. A good choice are reactive processors, as they do directly support preemption constructs and concurrency. Here the Kiel Esterel Processor (KEP) is used, as this reactive processor is based on the language Esterel. This processor also does not “optimize” the performance in such a way that it does not change execution statements to enhance execution time, threads have a strict order of execution. This and some other features facilitate WCRT analysis ([1], page 66). As the KEP is based on the Esterel language, the instruction set architecture (ISA) of the KEP is very similar to Esterel. Due to this direct mapping, most Esterel statements can be executed directly in just one instruction cycle. For those statements who cannot be performed directly, well-known translations exists, so that any Esterel program can be executed. Parts of the KEP ISA is shown in figure 3. The number of instruction cycles needed to execute a statement can be seen in the third column. This value is treated as the cost of a statement and will be used later to determine the WCRT of a program. Note that the pause statement is executed in both consecutive ticks and consumes an instruction cycle each time. Esterel’s parallel operator (∥) gets translated to KEP ISA like it is shown in the first row. For each thread, one PAR operand is needed (e. g. to set priority) and additionally one PARE to define the end of the last thread. Consequently, the cycle time needed to spawn n thread adds up to n+1. In addition to that, every time a thread terminates, the JOIN statement must be executed, which also consumes one instruction cycle. The KEP employs a multi-threaded architecture to implement this concurrency. Threads are scheduled according to their priority. This scheduling and context switching does not cost extra instruction cycles. The ABORT statement consumes two cycles only upon entering. No additional cycles are needed to test for an abort, as a watcher unit takes care whether a signal triggering this abort is present or not. This watcher unit is configured within these instruction cycles.

**The Concurrent KEP Assembler Graph (CKAG)** In order to simplify WCRT analysis and to make it more feasible, a new intermediate data structure is introduced here, called The Concurrent KEP Assembler Graph (CKAG). The CKAG is a graph structure consisting of nodes and edges and matches KEP program behaviour. It is built from Esterel source by traversing recursively over its Abstract Syntax Tree (AST) generated by the Columbia Esterel Compiler [3]. The nodes of the CKAG directly represent Esterel statements. The WCRT analysis is performed on the CKAG. The CKAG distinguishes between different types of edges to reach a successor. Given a node n:
Figure 3: Overview of the KEP instruction set architecture, and their relation to Esterel and the number of processor cycles for the execution of each instruction.

- $n.suc_c$ denotes all sequential control flow successors
- $n.suc_s$ and $n.suc_w$ lead to a successor reached via a strong or a weak abort
- $n.suc_e$ for exceptions

The CKAG also distinguishes between different types of nodes:

- **transient nodes**: represent instantaneous execution (e.g., `emit`) (displayed as a square box)
- **delay nodes**: represent statements that may hold for more than one tick (e.g., `pause`). When a delay node is created, additional preemption edges are added according to abortion context. (displayed as octagon)
- **fork and join nodes**: represent concurrency (triangles)

The CKAGs corresponding to ExSeq and ExPar can be found in figures 1(b) and 2(b), respectively. Note that all edges are plain sequential edges ($n.suc_c$), except the one in ExSeq leading from the pause statement ($n.suc_w$). It is annotated with a W.

### 3.1.2 Worst Case Reaction Time Analysis

As stated earlier, the instruction cycles of a given KEP program are used to determine the WCRT of this given program. The WCRT is defined as the maximum number of KEP cycles executable in one tick. Thus WCRT analysis requires finding the longest instantaneous path in the CKAG. If the program does not contain fork and join nodes (i.e., concurrency), a restricted algorithm can be used to determine WCRT of sequential programs, which can be understood more easily and therefore is introduced here before the general algorithm. This general algorithm further requires an analysis of instant reachability between fork and join nodes. The handling of concurrency is then added to the sequential WCRT algorithm, which is presented here now:

**Sequential WCRT Algorithm**

The basic idea of this algorithm is a Depth First Search traversal of the CKAG. From every node where execution may start in (e.g., pause, start), the WCRT is computed, as these are the nodes where the ticks of the program start. The overall WCRT of the program will then be the maximum of these calculated WCRTs. To do this, a value $n.inst$ is computed for each node $n$,
which gives the WCRT from this node on in the same instant when execution reaches the node. Here it is important to distinguish between the two types of nodes:

- For a **transient node**, the WCRT is simply the sum of the execution time of all its successors (children) plus its own execution time. This value equals \( n.\text{inst} \).

- For **delay nodes** (i.e., pause) it needs to be differentiated between two cases within a tick:
  
  - control can reach **delay node** \( d \) (pause), meaning that the thread who is executing \( d \) has already executed some other instructions in that tick. If execution continues within the same tick, meaning that the delay node does not delay (i.e., when the abort takes place), the WCRT from \( d \) on is expressed by the already introduced variable \( d.\text{inst} \). In the CKAG the \( \text{suc}_w \) or \( \text{suc}_s \) edge is used.
  
  - If execution starts in \( d \), meaning that \( d \) must have been reached in some preceding tick and it did delay at this point, an additional value \( d.\text{next} \) stores the WCRT from \( d \) on within the next tick. Here \( \text{suc}_s \) is used.

Referring to the example ExSeq in figure 1, figure 4(a) shows how the algorithm calculates the overall WCRT of the whole program. Each node \( n \) in the CKAG is annotated with a label \( W(n.\text{inst}) \), every delay node with a label ”\( W(n.\text{inst})/n.\text{next} \)”. Line numbers are annotated with \( L(n) \) in the CKAG and in the KEP assembler code. The algorithm starts with resetting all values, then computes \( n.\text{inst} \) for all nodes \( n \) and \( d.\text{next} \) for all delay nodes \( d \) and then returns the highest value found. It is \( g.\text{root} = L1 \).

1. The sequential WCRT computation starts initializing the \( \text{inst} \) and \( \text{next} \) values of all nodes to \( \bot \) (line 2 in getWCRTSeq(), figure 4(a))
2. Then `getInstSeq(L1)` is called. As L1 is a transient node, it computes:
   
   \[ L1.\text{inst} := \max \{ \text{getInstSeq}(L2) \} + \text{cycles}(\text{WABORT}_L1) \]  
   line 4 in `getInstSeq()`

3. The above statement `getInstSeq(L2)` calculates the instantaneous sequence for L2, which is a \textit{pause} statement and therefore a \textit{delay node}. Hence, L2 is computed as follows (line 6 in `getInstSeq()`):
   
   \begin{itemize}
   \item L2.\text{inst} := \text{cycles}(\text{PAUSE}_L2) + \text{cycles}(\text{EMIT}_L5) + \text{cycles}(\text{HALT}_L6) = 3
   \item Note that in the CKAG the \text{succ.w} edge annotated with a W is used
   \item Consequently, L1.\text{inst} := 3 + 2 = 5
   \end{itemize}

4. Back in `getWCRTSeq()` in line 4, the only delay node in the program is the \textit{pause} statement L2, therefore `getNextSeq(L2)` is called, which computes L2.\textit{next} := `getInstSeq(L3)` + `cycles(PAUSE)_L2`

5. The above statement `getInstSeq(L3)` computes and returns:
   
   \[ L3.\text{inst} := \text{cycles}(\text{EMIT}_L3) + \text{cycles}(\text{GOTO}_L4) + L2.\text{inst} = 1 + 1 + 3 = 5 \]

Therefore, as calculated in step 4 of the above example, the overall WCRT value of the program ExSeq is L2.\textit{next} := 5 + 1 = 6, which corresponds to the longest path in the CKAG, when the signal I becomes present. This takes place in the third tick, and the trace of the executed statements can be seen in figure [1](d). The abortion triggered by I is weak, the abort body is still executed in this instant, which takes four instructions: \textit{PAUSE}_L2, \textit{EMIT}_L3, the \textit{GOTO}_L4, and \textit{PAUSE}_L2 again. Then it is detected that the body has finished its execution for this instant, the abortion takes place, and \textit{EMIT}_L5 and \textit{HALT}_L6 are executed. Hence the longest possible path takes six instruction cycles. ([1], page 72).

**General WCRT Algorithm** The general algorithm is way more complicated than the sequential algorithm, because it has to be extended by several features. Their presence might not be of importance or it might not be clear in which way they are used when going through the example ExPar, but they are necessary to calculate the WCRT correctly. The algorithm proceeds as follows:

In a first step it is computed whether concurrent threads terminate instantaneously, thereafter it is possible to compute for each statement how many instructions are maximally executable from it in one logical tick. The maximal value over all nodes determines the WCRT of the program. ([1], page 77)

For the general WCRT algorithm it is important whether a \textit{fork} and its corresponding \textit{join node} can be executed within the same instant or whether a thread holds a delay node. The complete algorithm to do this is presented in detail in [2], it computes for a \textit{source} and a \textit{target} node whether the \textit{target} is reachable instantaneously form the \textit{source} ([1], page 74). The basic idea is to distinguish between three potential reachability properties for each node. In the WCRT algorithm only the properties of the \textit{fork} and \textit{join} are of interest.

- **instantaneous**: The \textit{join node} is reachable instantaneously form the corresponding \textit{fork node}, i.e. there is no \textit{delay node} within any of the spawned threads.

- **not-instantaneous**: At least one thread holds a \textit{delay node}, meaning that the \textit{join node} is not reached within the same tick.

- **exit-instantaneous**: in case of an exception within a thread, control jumps to the exception handler, but on the KEP the \textit{join node} gets executed to terminate the thread correctly. If a \textit{join node} is executed this way, the statements that are instantaneously reachable from it are not executed, but control instead moves on to the exception handler ([1], page 74). This behaviour is expressed in the \textit{exit-instantaneous} property.
As stated earlier, the general algorithm extracts form the sequential one, but is extended with some more extras. These further additional extras are of interest and need to be explained in detail:

- For a **fork node** the \( f.\text{inst} \) value is simply the sum of the instantaneously reachable statements of its sub-threads, plus the \( \text{PAR} \) statement for each sub-thread and the additional \( \text{PARE} \) statement ([1], page 75). If the corresponding **join node** is instantaneous reachable, the \( f.\text{inst} \) value also includes all further statements. Referring to the example ExPar \( ^2 \) the \( f.\text{inst} \) value of the **fork node** \( L_3 \) is computed as follows:
  \[
  L_3.\text{inst} = 2 \times \text{cycles(PAR)} + \text{cycles(PARE)} + L_4.\text{inst} + L_5.\text{inst} = 2 + 1 + 2 + 2 = 7.
  \]
  Note that the **fork/join** pair is always **non-instantaneous**, due to the \( \text{PAUSE}_{L_6} \) statement.

- For a **join node** \( j \) the \( j.\text{inst} \) value is also very easy to compute, as it is computed in the same way as all \( \text{inst} \) values. In the example ExPar computation of the **join node** \( L_8.\text{inst} \) value is done like this:
  \[
  L_8.\text{inst} = L_9.\text{inst} + \text{cycles}() = L_3.\text{inst} + \text{cycles}(\text{GOTO}_{L_9}) + \text{cycles}(\text{JOIN}_{L_8}) = 7 + 1 + 1 = 9
  \]

- Also, the **join nodes**, like **delay nodes**, have a **next** value. When a **fork-join pair** \((f, j)\) is **non-instantaneous**, meaning that at least one thread delayed and the **join node** is not executed within the same tick, the **next** value holds the next instants analogously to the **delay nodes**. Its computation requires first the computation of the **next** value of the **delay nodes** of all threads:
  \[
  L_6.\text{next} = \text{cycles}(\text{PAUSE}_{L_6}) + \text{cycles}(\text{EMIT}_{L_7}) = 2
  \]
  The join **next** value is then the sum of the **next** value of each thread plus the \( \text{WCRT} \) of the successors, i.e. the **inst value**. In the example the **next** value of the **join node** \( L_8 \) is computed as follows:
  \[
  L_8.\text{next} = L_6.\text{next} + L_8.\text{inst} = 2 + 9 = 11
  \]
  ([1], page 76)
  The overall WCRT of the example ExPar is 11, the trace of the executed statements can be seen in figure\( ^2 \)(d).

### 3.2 Model Checking Approach

The method presented in [3] is called the max-plus approach, because it calculates the maximum local tick length of each tick and then adds them up to gain the overall WCRT value. From now on, the WCRT calculated this way will be called \( WCRT_{\max} \), as this is the upper bound of the WCRT and usually a gross overestimation above the real WCRT value. Accordingly, \( WCRT_{\min} \) is defined as the sum of the minimum local tick lengths. The real WCRT value lies between these two, i.e. in the interval \([WCRT_{\min}, WCRT_{\max}]\).

The approach presented now is about finding a tighter value than \( WCRT_{\max} \), which usually holds the fact that any value less than this value may cause a timing fault during the execution. This value is referred to as \( WCRT_{\text{tight}} \) ([4], page 209). An example to these values can be found at the TFSMs in section 3.2.2 later.

For this approach the synchronous language PRET-C is used, which is a synchronous extension to C. This language, together with an example, is presented in the following section. Then, the program gets translated to a new intermediate format called Timed Concurrent Control Flow Graph (TCCFG). This is generated from the assembler level and gives precise values for each tick in terms of clock cycles (The processor used is called ARPRET instead of the KEP from the maxplus approach) ([4], page 208). This TCCFG is reduced to TFSMs for each thread of the program, to display relevant information only. From there on a Timed Automata (TA) model is build to finally perform the WCRT analysis.
```c
#include "pretc.h"
#define N 1000
go void sampler ( void ) ;
go void display ( void ) ;
EXTERN sensor ;
int cnt = 0 ;
float buffer [ N ] ;
go int main ()
  { PAR ( sampler , display ) ;
    return 0 ;
  }
go void sampler ()
  { int i = 0 ;
    float sample ;
    while ( 1 )
      { sample = read ( sensor ) ;
        EOT ;
        while ( cnt == N )
          EOT ;
        buffer [ i ] = sample ;
        EOT ;
        i = ( i + 1 ) % N
        cnt = cnt + 1 ;
      }
  }
go void display ()
  { int i = 0 ;
    float out ;
    while ( 1 )
      { EOT ;
        while ( cnt == 0 )
          EOT ;
        out = buffer [ i ] ;
        EOT ;
        i = ( i + 1 ) % N
        cnt = cnt - 1 ;
      } WriteLCD ( out ) ;
  }
```

Figure 5: A PRET-C example: consumer producer

### 3.2.1 PRET-C and the TCCFG

**PRET-C Overview** Precision Timed C (PRET-C) is a synchronous extension of the C language, which is specially designed for predictable execution on the used processor and is thread safe by construction. Standard shared variables are used and three major constructions are added to C ([4], page 207):

- **PAR(T,U)** spawns two synchronous parallel threads T and U to emulate concurrency. Due to the textual order, T has higher priority over U and therefore is scheduled first in each tick.

- **EOT** Similar to the pause statement in Esterel, when an EOT statement is reached, the thread completes its local tick. A global tick elapses only when all participating threads of a PAR() reach their respective EOT. Precise timing of execution is ensured, because the next global tick is started only when all threads have reached their EOT.

- **[weak] abort.** This abort construct works like the one from Esterel. In case of a strong abort, the preemption happens at the beginning of the instant and preempts the body there, weak abort allows its body to execute one last time.

Further Details of the language and its semantics are presented in [5]. More detailed explanation of the above statements can be found in [4] at pages 207-208.

A PRET-C example can be found in figure 5. The main thread spawns two threads, which are basically normal C functions. The sampler thread reads data from a sensor and deposits these on a global circular buffer, the display thread reads the data from the buffer and displays them. Communication occurs via the shared variables cnt and buffer, the sampler thread has a higher priority over the display thread due to the textual order.

In this program, the sampler thread starts by reading the sensor data in its first local instance of time (local tick). In the next instant, it checks if the data buffer is full, and in this event it just ends its local tick. As long as the buffer is full, it keeps on waiting until the display thread has read some data so that empty space is available. If it successfully comes out of the while loop, it writes to the next available location of the buffer and ends another local tick. In the next instant of time, the index of the buffer
and the total number of data in the buffer are incremented (note that this is a circular buffer). Then the 
sampling loop is restarted.

The display thread starts by first checking if there is any data available to be read ($\textit{cnt} \neq 0$). If there is 
no data available, the thread ends its local tick and keeps on waiting until some data is deposited by the 
producer. When this happens, it reads the next data from the buffer and ends its local tick. In the next 
instance, the value of $\textit{cnt}$ is decremented and in the final instance the data read is sent to a display device.

The ARPRET processor makes sure that the concurrency is performed correctly in order of the fixed 
scheduling of the threads and therefore no race conditions are possible (\cite{4} page 208).

**Timed Concurrent Control Flow Graph (TCCFG)** The TCCFG is an intermediate graphical format 
which decodes the explicit control-flow of the threads and also has information regarding forking and 
joining of the threads. It is generated from the assembler level so as to get precise values for each 
instant in terms of ARPRET clock cycles (\cite{4} page 209). The exact clock cycles needed to execute 
the instruction can be found beside each node in the TCCFG. The according TCCFG to the Producer 
Consumer example can be found in figure 6(a).

The following types of nodes can be found in the TCCFG:

- **Start/End node:** Every TCCFG has a unique start node where the control begins and may have 
an end node, if the program can terminate.

- **Fork/Join nodes:** These are needed to clearly mark concurrent threads of control and where these 
threads start and end.

- **Action nodes:** These are used for any C function call or data computation.
• EOT nodes: These nodes indicate a local end of tick.

• Control flow nodes: There are two types of control flow nodes: conditional nodes to implement conditional branching and jump nodes for mapping unconditional branches (which are needed to emulate infinite loops).

3.2.2 WCRT Analysis using Model Checking

Thread Finite State Machines (TFSMs) The WCRT analysis is performed on an equivalent model, the Timed Automata (TA). For illustration purposes, the TCCFG is mapped to TFSMs first, which are then mapped to an equivalent TA. The TFSMs corresponding to the two threads of the producer consumer TCCFG \( a \) is shown in figure 6(b) and 6(c). This mapping is done automatically by a depth first search from every EOT node to all EOT nodes that are reachable from this node. All other nodes are omitted. During the traversal, the cost of every node is simply added to obtain the total cost between these two EOTs ([4], page 210). The calculated costs are then annotated at the transitions between the EOT nodes. The composition between TFSMs is strictly synchronous. Using these TFSMs, it is quite easy to calculate the earlier introduced values \( WCRT_{max} \) and \( WCRT_{min} \). The maximum local tick length of the Sampler thread is 54 and 56 of the Display thread, hence \( WCRT_{max} = 54 + 56 = 110 \). Equally, \( WCRT_{min} = 31 + 28 = 59 \), therefore the value of \( WCRT_{tight} \) is within the interval \([59,110]\) ([4], page 210). Later it can be seen that the two threads with the highest local tick length are never executed within a global tick and therefore \( WCRT_{tight} \) is below \( WCRT_{max} \).

Timed Automata (TAs) As this approach is called the model checking approach, one needs a formal model and a property which is checked by the model checker. Here the \( WCRT_{tight} \) computation problem is modelled to the checking of a CTL property in the model checker UPPAAL[6] over an automata with integer variables but no clocks. Model-Checking is a verification method of systems to prove or refute correctness of a system regarding its specifications or properties[8]. UPPAAL is a state-of-the-art model checker for modelling, validation and verification of real-time systems modelled as networks of timed automata (TA)[7]. The property of the producer consumer example which the TA is tested against is formulated in CTL, namely \( AG(gtick \Rightarrow x \leq val) \). This is explained in detail later.

This paragraph is about the mapping of the TFSM to a timed automata. In order to preserve synchrony, this mapping has to be done carefully, because the compositions of the TA are asynchronous. Each TFSM gets mapped to an equivalent TA. An additional TA, called a barrier, is also introduced to realize the synchronous semantics of PRET-C execution.

The TA of the producer consumer example is shown in figure 7. In the TA, there are two kinds of states:

• EOT states: Each \( EOT_i \) form the TFSMs equals an identical state \( EOT_i \) in the TA.

• barrier states: Every transition \( [EOT_i] \xrightarrow{d} [EOT_j] \) in the TFSM is replaced in the TA by two transitions by adding a new state in between, called a barrier state \( b_{ij} \).

Each transition has two parts:

• the upper part is the transition guard. It is the enabling condition of the transition (e. g. \(!gtick\)). Only if this condition is true, the transition is taken.

• the lower part of a transition is an action that is executed when the transition is taken (e. g. \( x=x+54, \; lt1=true \) at the transition from \( EOT0 \) to \( EOT1 \)).
The variable \( x \) is used to capture the cost of each global tick. The boolean variable \( lt \) captures whether a given thread has completed its local tick. The boolean variable \( gtick \) is true when the global tick has happened.

Referring to the first tick of the Sampler thread in Figure 7(a), the transition from EOT0 to the barrier node \( b01 \) is taken when the global tick hasn’t happened (this is the transition guard \( !gtick \)). While taking this transition, the variable \( x \) is incremented by the cost of the transition (54) and the local tick \( (lt1) \) is set to true. Then the automaton reaches the barrier node and stays there until the global tick happens. The task of ensuring that the barrier has been reached is handled by a third automata called the barrier as shown in Figure 7(c). The barrier has just two states called WaitLT and GTReached. The barrier remains in the WaitLT state until both \( lt1 \) and \( lt2 \) have been set to true by the two threads. In this case, both TAs reached their respective barrier state. The barrier TA will then set the global tick variable \( gtick \) to true and will wait in the state GTReached. The barrier TA resets back to the initial state only when both automata have completed their respective barrier transitions, in response \( gtick \) becomes false. Note that when the transition to the initial state is taken by the barrier TA, the value of the counter \( x \) is reset. Thus, when the state GTReached is reached, the value of \( x \) captures the cost of a global tick of the program (4, page 210-211).

**WCRT as a CTL Property** The CTL (Computational tree logic) is an extension to the Propositional calculus to model certain relationships for a logical proposition \( \varphi \), e. g. “\( \varphi \) is always true” or “\( \varphi \) gets true in the next instance” (5 page 5). This is done by so called **temporal Operators**. To express that a property of a system must apply, the **path quantifier** \( A \) is used. The quantifier \( G \) means “globally”, i. e. \( \varphi \) is valid in every state on all transitions (6 page 6). The formula \( AG(gtick \Rightarrow x \leq val) \), which is entered into the model checker, means: On all transitions in all states is \( (gtick \Rightarrow x \leq val) \) valid.

The variable \( val \) is a value entered by the user, which of course must lay in the interval \([WCRT_{min}, WCRT_{max}]\), too. Basically, the first value of \( val \) entered in the model checker is a guess. The model
checker then tells whether this guess is equal to or less than the WCRT of the system. In order to not check every value in the interval $[WCRT_{\text{min}}, WCRT_{\text{max}}]$, standard binary search can be used to minimize the number of queries. In case of the producer consumer example, this interval is $[59, 110]$ and therefore at most 6 values have to be entered in the model checker ($\log_2 (110-59) = \log_2 (51) = 6$). In the producer consumer example, the tight value $WCRT_{\text{tight}}$ obtained by the above analysis is 92 in comparison to the maximum value of 110 ([4], page 211).

3.3 Reachability Approach

In this approach the WCRT computation problem is solved by on-the-fly reachability analysis ([9], page 481). Similar to the model checking approach, PRET-C and the TCCFG are used to describe the program and to extract information about timing and concurrency. To simplify explanation, the same producer consumer example from the previous approach is used. The code and the TCCFG can be found in figure 5 and 6, respectively.

In case of an error state, a thread terminates. This is not implemented in the abstracted PRET-C code in figure 5, but it is important for termination of the reachability analysis. It can be added easily to the code by introducing error states.

3.3.1 WCRT analysis using Reachability

In figure 8(a), the exact same FSMs of the two thread are shown, as they where introduced in section 3.2.2 and figure 6(a) & (b), but displayed as a more accurate translation from the TCCFG. Each individual path in the TCCFG 6(a) starting from the Fork node gets translated into a separate thread FSM. The original Fork node is left with a single transition to the corresponding Join node (node 2 → 3). This transition fires when all children threads terminate ([9], page 482).

Reachability Algorithm The algorithm uses PRET-C and the synchronous composition of the TFSMs to compute the reachable states. A global variable WCRT is used, which contains the highest reaction time visited at any stage during reachability analysis ([9], page 482). As explained earlier, threads spawned by the PAR construct have a fixed order of execution. In figure 5 line 7, PAR(sampler, display) implies that sampler executes before display in every tick. Also, the spawning thread (main) is suspended until all spawned threads terminate.

Consider figure 8(b) which shows a part of the forward reachability computation for the three abstracted threads shown in figure 8(a). The initial state of the reachability graph relates to the initial node (node 1) of the main thread (main). The reachable state space is traversed in a depth-first fashion. Whenever a transition to a fork node is seen (node 1 to 2 in main), the initial nodes (5 and 6) of the newly spawned threads (sampler and display) are added to the state tuple. The spawning thread node (node 2) is suspended and must wait for the spawned threads to finish. From (2,5,6), the two spawned threads make transitions to EOT nodes 7 and 8 respectively, marking the end of the first global tick. Next, both threads take transitions to nodes 9 and 10 respectively, marking the end of the second tick. Now, from (2,9,10), there are four possible transitions. Transitions to (2,11,12), (2,13,12) and (2,13,14) lead to loops which would extend the analysis too much and therefore are not illustrated further. The fourth successor (2,11,14) may lead the (2,13,16) and then to (2,15,17), which represents the termination of both spawned threads, which re-enables the fork node 2 to take its lone transition to the join node 3 within the same tick. Then transition to node 4 is taken, which marks the end of the last global tick ([9], page 482).
(a) Translation of TCCFG into FSMs
(b) Reachability illustration

Figure 8: FSMs and the Reachability illustration

Page 483). Note that from the tuples in tick 3 and 4 there are other transition possible which are not shown in the illustration (b).

A previously visited state is never visited again, therefore termination of the reachability analysis is guaranteed. This is also the reason for introducing the error states earlier, so that the analysis can terminate. During this reachability computation, when a global tick finishes, the global variable WCRT (initialized to 0) is compared to the computed tick length. If WCRT is smaller than the tick length (sum of the cost of the transitions taken), the value is updated. Otherwise, it is left unchanged. On termination, WCRT contains the actual WCRT of the program ([9], page 483).

For example, in tick 2, WCRT = 54+29 = 83 and in tick 3 it is 31+31 = 62, therefore WCRT does not get updated. During reachability, it is not necessary that every EOT state in a thread is combined with every EOT state in a parallel thread. Only reachable states are analysed ([9], page 484).

The calculated WCRT value by the reachability approach does not differ from the result gained form model checking, but it reduces the time needed to calculate WCRT significantly.

4 Complexity and Results

Complexity In this section the complexity and time consume of the different approaches are presented. Furthermore, a set of experimental evaluation of several programs are submitted.

Regarding the complexity of the general algorithm of the Max-Plus approach, its complexity is fairly easy to calculate: Let $n := |Nodes|, d := |DelayNodes|, f := |ForkNodes| and j := |JoinNodes|$. For each node its WCRT’s inst is computed at most once, additionally the next has to be computed for the delay and join nodes. For all fork nodes a fork-join reachability analysis is additionally made, which has itself $O(n)$. Altogether the complexity is $O(n+d+j) + O(f\times n) = O(2\times n) + O(n^2) = O(n^2)$ ([1], page 76).

Determining the complexity of the model checking approach and of the reachability approach is not as straightforward as for max-plus. Therefore, a detailed explanation is not given here, but rather can be
read in [4] in section 3.6. Regarding the Model Checking approach, the use of binary search means that the model checking has to be carried out in multiple steps. The complexity lays therefore at Product of thread sizes × binary search. Reachability computes WCRT on-the-fly and therefore does not consume time for several steps. Its complexity is just the Product of thread sizes ([9], page 480).

**Results** In order to compare the different approaches against each other, the developers came up with some examples to present their results. In figure 9 are four examples shown, which feature different amounts of threads and lines of codes ([9], page 485). The fourth column shows the WCRT\textsubscript{max} obtained by the Max-Plus approach, the fifth column shows the WCRT\textsubscript{tight} gained by Model Checking. Please note that Reachability does not provide tighter results. The overall gain of WCRT\textsubscript{tight} over WCRT\textsubscript{max} is on average 10.72%, (if more examples are evaluated). The advantage of the Reachability approach over model checking can be seen in the last two columns, where the time needed to calculate the WCRT is shown. It becomes quite obvious that the Reachability approach is a significant speed up compared to the Model Checking approach.

### Table 1: Comparison of the three approaches

<table>
<thead>
<tr>
<th>Example</th>
<th>Threads</th>
<th>LOC</th>
<th>WCRT\textsubscript{max}</th>
<th>WCRT\textsubscript{tight}</th>
<th>Gain (%)</th>
<th>MC time (ms)</th>
<th>R time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer-Consumer</td>
<td>2</td>
<td>567</td>
<td>110</td>
<td>92</td>
<td>16.36</td>
<td>157</td>
<td>1</td>
</tr>
<tr>
<td>Smokers</td>
<td>4</td>
<td>648</td>
<td>531</td>
<td>449</td>
<td>15.44</td>
<td>297</td>
<td>1</td>
</tr>
<tr>
<td>Robot Sonar</td>
<td>4</td>
<td>1081</td>
<td>419</td>
<td>346</td>
<td>17.42</td>
<td>9407</td>
<td>13</td>
</tr>
<tr>
<td>Channel Protocol</td>
<td>4</td>
<td>727</td>
<td>174</td>
<td>152</td>
<td>12.64</td>
<td>969</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
<th>Threads</th>
<th>LOC</th>
<th>WCRT\textsubscript{max}</th>
<th>WCRT\textsubscript{tight}</th>
<th>Gain (%)</th>
<th>MC time (ms)</th>
<th>R time (ms)</th>
</tr>
</thead>
</table>

**5 Conclusions**

In this report I summarized three different approaches to compute the worst case reaction time, or WCRT, of synchronous programs. The analysis was done using two different languages, namely Esterel and PRET-C, which are constructed especially for the needs of reactive systems, e. g. to support multi-threading and thread safe access to shared variables. Then, the programs are transformed to different intermediate data structures where the WCRT analysis is then performed. The first approach supplies a fairly easy solution, but the result is usually an overestimation and therefore useless. The Model Checking approach lowered the result to a useful value, but its computing time is very high. The third approach reachability has lower complexity, but still achieves the same tightness in WCRT computation as the other more complex techniques.

### Bibliography

**References**


