# Timing analysis of synchronous programs using WCRT Algebra: Scalability through abstraction

# Timing analysis of synchronous programs using WCRT Algebra: Scalability through abstraction

JIAJIE WANG, University of Auckland
MICHAEL MENDLER*, University of Bamberg
PARTHA ROOP†, University of Auckland
BRUNO BODIN, University of Edinburgh

Synchronous languages are ideal for designing safety-critical systems. Static Worst-Case Reaction Time (WCRT) analysis is an essential component in the design flow that ensures the real-time requirements are met. There are a few approaches for WCRT analysis, and the most versatile of all is explicit path enumeration. However, as synchronous programs are highly concurrent, techniques based on this approach, such as model checking, suffer from state explosion as the number of threads increases. One observation on this problem is that these existing techniques analyse the program by enumerating a functionally equivalent automaton while WCRT is a non-functional property. This mismatch potentially causes algorithm-induced state explosion. In this paper, we propose a WCRT analysis technique based on the notion of timing equivalence, expressed using WCRT algebra. WCRT algebra can effectively capture the timing behaviour of a synchronous program by converting its intermediate representation Timed Concurrent Control Flow Graph (TCCFG) into a Tick Cost Automaton (TCA), a minimal automaton that is timing equivalent to the original program. Then the WCRT is computed over the TCA. We have implemented our approach and benchmarked it against state-of-the-art WCRT analysis techniques. The results show that the WCRT algebra is 3.5 times faster on average than the fastest published technique.

Additional Key Words and Phrases: WCRT analysis, synchronous languages, timing algebra

**ACM Reference format:**
JiaJie Wang, Michael Mendler, Partha Roop, and Bruno Bodin. 2017. Timing analysis of synchronous programs using WCRT Algebra: Scalability through abstraction. *ACM Trans. Embedd. Comput. Syst.* 0, 0, Article 0 ( 2017), 21 pages.
https://doi.org/0000001.0000001

*The author is supported under grant PRETSY2 by the German Research Foundation DFG-1427/6-2.
†The author was supported under under grant PRETSY2 by the German Research Foundation DFG-1427/6-2 and research and study leave from Auckland University

This article was presented in the International Conference EMSOFT 2017 and appears as part of the ESWEEK-TECS special issue.

1539-9087/2017/0-ART0 $15.00
https://doi.org/0000001.0000001

## 1 INTRODUCTION

Safety-critical embedded systems require both functional and timing correctness. Synchronous languages have been a paradigm of choice for the design of such applications, particularly in aviation. This paper considers the recently developed Precision Timed C (PRET-C) language [5], which combines a subset of C with synchronous constructs, in order to guarantee determinism, reactivity, and thread-safe communications.

In synchronous languages, time progresses in logical discrete instants called *ticks*. These identify synchronisation points between threads and events. In each tick, the environment is sampled at the beginning, then the computation takes place, and the results are emitted at the end. Thus, the length of a tick defines the system's reaction time to external events. To guarantee that a system will always react on time, we need to ensure this reaction time is short relative to the environment. This necessitates the computation of the worst-case tick length, also known as Worst-Case Reaction Time (WCRT) analysis.

WCRT analysis is no trivial task. The most commonly used approach is *implicit path enumeration* [8, 11, 16]. This approach abstracts explicit program execution as sets, which subsequently enables the program to be modelled using Integer Linear Programming (ILP). This abstraction greatly enhances the scalability of the approach. However, it is also conventionally known to be less precise than other approaches.

In contrast, *explicit path enumeration* preserves the semantics of programs during the analysis. The advantage of this approach is that it can be easily extended for value tracking, thus computing a precise WCRT, which is very difficult to achieve with implicit path enumeration. All existing model checking and reachability based techniques follow this approach [4, 10]. However, with explicit path enumeration there is a high risk of state explosion when concurrent threads are considered.

There is also a hybrid approach which attempts to strike a balance between implicit and explicit path enumeration [9, 13]. The idea is to use ILP as a baseline and use model checking [13] or reachability [9] analysis to enhance the precision. However, the results are not very promising since the analysis time is mostly dominated by the explicit path enumeration.

Apart from actual techniques, there are also theories developed for WCRT analysis. Recently a timed semantics of synchronous programs is developed in [12] for the timing analysis of an intermediate-level representation of the SCCharts language [15], called input-output boolean tick cost automata (IO-BTCA). This theoretical work addresses the use of min-max-plus algebra for obtaining sound abstractions of IO-BTCA, ranging from modelling of exact signal-dependent behaviour to fully signal-abstract models. However, [12] only treats flat parallel compositions of IO-BTCAs. It cannot handle hierarchy and preemption. Furthermore, it also lacks methodologies for transforming between the Control Flow Graph (CFG) and IO-BTCAs, which is necessary for an actual implementation.

In this paper, we propose a novel and practical WCRT analysis technique which combines the benefits of the explicit path enumeration approach with those of the algebraic setting. Existing explicit path enumeration techniques compute the WCRT by first constructing a functionally equivalent automaton from the program, and then enumerating its states [4, 9, 10]. While this is intuitive, it suffers from the state explosion problem. In the proposed technique, we exploit the abstraction power of timing equivalence in min-max-plus algebra in order to reduce the effects of state space explosion significantly.

The contributions of this work are the following:

- We propose *Tick Cost Automata* (TCA) for WCRT analysis based on the idea of *timing equivalence* and formalise their behavioural semantics using min-max-plus algebra, which we term *WCRT algebra*. This provides a sound and more direct mathematical formulation of WCRT analysis.
- We develop a transformation of Timed Concurrent Control Flow Graph (TCCFG) into Tick-Cost Automaton (TCA), which allows the TCA to be generated in a fully structural manner along the hierarchy of the program.
- We implemented a timing simulation based on the WCRT algebra and experimentally compared its performance with the state-of-the art model checking [4] and ILP based [16] WCRT analysis techniques. On our benchmarks, the results show that our implementation of WCRT algebra is as precise as the other techniques whilst being considerably faster.

The remainder of this paper is organised as follows: We first introduce a motivating example in Sec. 2, then present a brief overview of the intermediate format used for analysis, called TCCFG, in Sec. 3. Following that, we show our formalisations of TCA and the proposed WCRT algebra in Sec. 4. The experimental results are reported in Sec. 6. Finally, we make concluding remarks in Sec. 7.

## 2 MOTIVATION

The idea of WCRT analysis is to model the program mathematically; in our case, using automata. For the same program, the automata can be different depending on the perspective of the modelling. Since WCRT is a timing property, we model a program from a timing perspective. The execution of a synchronous program can be considered as a sequence of ticks $\{Tick1, Tick2, Tick3, \dots\}$, and in each tick $i$ the program has a set of possible execution times $\{C_{Ticki\_1}, C_{Ticki\_2}, \dots\}$ corresponding to different branches in the program. The maximum execution time of tick $i$ can be defined as $C_{Ticki} = \max\{C_{Ticki\_1}, C_{Ticki\_2}, \dots\}$, and the WCRT is the largest execution time of all the ticks WCRT $= \max\{C_{Tick1}, C_{Tick2}, C_{Tick3}, \dots\}$. When programs are executed in parallel, assuming an interleaved (multi-threaded) execution, the timing costs are added to give the cost of a tick for the composite program. The fact that the maximum over sums is not the same as the sum over the maxima creates the crucial precision vs efficiency trade-off that we propose to handle using TCAs. In this section, we will illustrate the idea of TCA using a motivating example and compare it with the existing techniques.

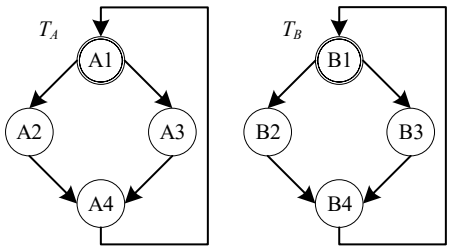


Fig. 1. Motivating example for comparing the proposed and the existing techniques.

Let us consider a synchronous program with two concurrent threads $T_A \| T_B$ as shown in Fig. 1. For simplicity, the two threads are identical in structure. Each thread has four states, with the initial states *A1* and *B1* respectively. In each tick, threads take exactly one

transition, then wait for each other. The WCRT of this program is the maximal possible timing cost in any reachable tick in the execution of $T_A \| T_B$. We write this timing value as $\mathrm{WCRT}(T_A \| T_B)$.

For Fig. 1 we can break down the computation of WCRT using the operators maximum $\oplus$ and addition $\odot$, where $X \oplus Y = \max(X, Y)$ and $X \odot Y = X + Y$. The operator $\odot$ can distribute over $\oplus$, *e. g.,* $X \odot (Y \oplus Z) = X + \max(Y, Z) = \max(X + Y, X + Z) = (X \odot Y) \oplus (X \odot Z)$. However, $\oplus$ cannot distribute over $\odot$. *E. g.,* $15 \oplus (5 \odot 10) = 15$ and $(15 \oplus 5) \odot (15 \oplus 10) = 30$. This is known as min-max-plus algebra, which will be presented in more detail in Sec. 4.

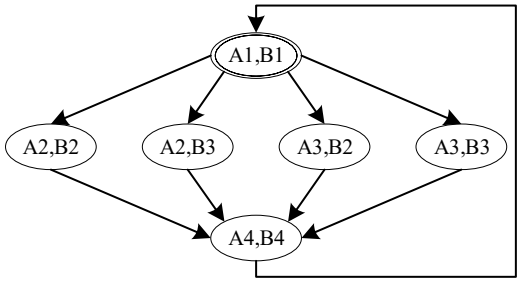


Fig. 2. Existing technique flatten the functional composition to obtain a single automaton.

• *Existing techniques based on explicit path enumeration.* The existing techniques compute the WCRT by flattening the functional composition into a single automaton $R \cong T_A \| T_B$ and then computing $\mathrm{WCRT}(R)$ from it. This flattened automaton $R$ is shown in Fig. 2. It captures all the possible reachable execution states. Initially, the program executes *A1* and *B1* concurrently. Thereafter, and the program reaches one of four possible states. Finally, the program executes *A4* and *B4*, and repeats. Computing $\mathrm{WCRT}(R)$ is to find the reachable program state which has the longest execution time:

$$\mathrm{WCRT} = (A1 \odot B1) \oplus (A2 \odot B2) \oplus (A2 \odot B3) \oplus (A3 \odot B2) \oplus (A3 \odot B3) \oplus (A4 \odot B4). \tag{1}$$

For this example, existing techniques require 11 operations to compute the WCRT, and the number of operations increases exponentially with the number of threads. This is a well-known aspect of explicit path enumeration. If we extend the motivating example with $n$ threads of identical structure, the number of required operations is $n2^n + 2n - 1$.

• *The existing max-plus approach.* The state explosion is a result of the concurrent composition of threads. This can be mitigated by abstracting the threads before composing them. One well-known technique is the max-plus approach [6], which abstracts the threads into a single value before composition, i. e., estimating $\mathrm{WCRT}(T_A \| T_B)$ as $\mathrm{WCRT}(T_A) + \mathrm{WCRT}(T_B)$. The computation is:

$$\mathrm{WCRT} = (A1 \oplus A2 \oplus A3 \oplus A4) \odot (B1 \oplus B2 \oplus B3 \oplus B4). \tag{2}$$

While this abstraction greatly reduces the number of operations ($4n - 1$ for $n$ threads), the modelling accuracy is also reduced. For example, if $\mathrm{WCRT}(T_A) = A1$ and $\mathrm{WCRT}(T_B) = B2$, then $\mathrm{WCRT}(T_A \| T_B)$ is estimated as $A1 \odot B2$. As shown in Fig. 2, this combination of states is infeasible. Thus, it is an overestimate of $\mathrm{WCRT}(T_A \| T_B)$. This phenomenon of infeasible states induced by the program structure is known as the *tick alignment problem* [4].

• *The proposed technique using TCAs.* It is a good idea to abstract threads before composition, however, the problem is how to preserve sufficient tick alignment to determine the exact value of WCRT($T_A \| T_B$). We propose to achieve this by considering the execution of a thread from a timing perspective.

An automaton is *WCRT equivalent* to the original program if it produces exactly the same timing sequence $\{C_{Tick1}, C_{Tick2}, C_{Tick3}, \dots\}$. Therefore, we can construct a minimal WCRT equivalent automaton of the program, and compute the WCRT from it. We call this automaton a *Tick-Cost Automaton (TCA)*. A TCA representation can be smaller and simpler than the control flow of the program, since only one state is needed for each tick, therefore the computation of WCRT can be much quicker.

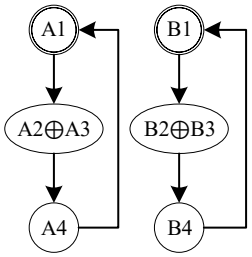


Fig. 3. Converting $T_A$ and $T_B$ into TCAs, and composing them to compute the WCRT.

The TCA of a thread can be derived using simple algebra. Fig. 3 shows the TCAs of the threads $T_A$ and $T_B$. Each TCA has three states, with costs shown inside. If we compose these two TCA using state-wise additions, we obtain the TCA of the program, which also has three states. The WCRT computation using TCA is as follows:

| | $\boldsymbol{T_A}$ | $\boldsymbol{T_B}$ | $\boldsymbol{T_A \parallel T_B}$ |
|---|---|---|---|
| Tick 1 | $A1$ | $B1$ | $A1 \odot B1$ |
| Tick 2 | $A2 \oplus A3$ | $B2 \oplus B3$ | $(A2 \oplus A3) \odot (B2 \oplus B3)$ |
| Tick 3 | $A4$ | $B4$ | $A4 \odot B4$ |
| ... | *(Repeat)* | *(Repeat)* | *(Repeat)* |

$$\text{WCRT} = (A1 \odot B1) \oplus ((A2 \oplus A3) \odot (B2 \oplus B3)) \oplus (A4 \odot B4). \tag{3}$$

By analysing the program from a timing perspective, we observe that the timing repeats itself every three ticks. The proposed technique can compute the WCRT from the TCA using only seven operations. More importantly, compared to the existing techniques, we do not lose any precision. If we expand the parentheses of the second term in Computation (3) using the distribution law, we obtain the exact same value as Computation (1). Finally, if we extend the example with $n$ threads of identical structure, the number of operations is $4n - 1$, which is the same as Computation (2).

## 2.1 Limitations

The motivating example demonstrates how the proposed technique can be as fast as the max-plus approach while being mathematically equivalent to the more precise explicit path enumeration. However, it has two limitations. First, the efficient composition of concurrent threads relies on abstracting signals. Signals are a common mechanism used in

many synchronous languages for communication between threads. Most existing explicit path enumeration techniques are capable of modelling signals. At this stage, the proposed technique falls short in this regard. However, even with signal abstraction, the tick alignment problem is still challenging enough for the existing technique to achieve practical analysis time, as shown in [16] and by our benchmarking in Sec. 6.

The second limitation is that the proposed technique does not solve the state explosion problem for all cases. If two TCAs cycle through a prime number of states, their composition is the multiplication of the two prime numbers, which will result in the same number of operations as the existing techniques. For example, consider two TCAs with three and five states respectively. In this case, this composed TCA will have 15 states, i. e., every state of $TCA_1$ will align with every other states in $TCA_2$. Hence, the number of the required operations will be equal to that of the existing techniques. However, we believe such cases are rare in normal programs, thus we can benefit from the abstraction most of the time.

```
ReactiveInput(int,In1,0);
ReactiveInput(int,In2,0);

void T1(){
    EOT;
    foo_B8();
}
void T2(){
    do{
        EOT;
        foo_B10();
        EOT;
    } while(In1);
}

void main(){
    // Box_abort
    abort{
        EOT;
        // Box_fork
        PAR(T1, T2);
    } when(In2);
    EOT;
    foo_B16();
}
```

Fig. 4. A PRET-C example.

## 3 TIMED CONCURRENT CONTROL FLOW GRAPH

WCRT algebra is formulated for PRET-C [5] and its intermediate format Timed Concurrent Control Flow Graph (TCCFG) [14]. Fig. 4 shows an example of a PRET-C program. PRET-C extends the C language with three major synchronous statements: The *End Of Tick (EOT)* statement which marks the state boundaries, the *PAR* statement for spawning concurrent threads and the *abort* statement for preemption. The main thread in Fig. 4 consists of a strong abort. When the environment input `In2` is true, everything enclosed inside the abort body is preempted immediately. The abort itself terminates when both *T1* and *T2* terminate (i. e., join) or when the preemption `In2` takes place.

The TCCFG in Fig. 5 reflects the control flow of the PRET-C program of Fig. 4, with annotated timing information. A TCCFG has the following types of nodes: conventional *start*, *end*, *computation* and *condition* nodes, with additional *abort-start* and *abort-end* nodes for preemption, *fork* and *join* nodes for concurrency, and *EOT* nodes for the pauses (i. e., state boundaries). Each node is annotated with an execution cost in processor clock cycles. In our case, these costs are derived using the technique presented in [7]. The intuitive semantics of PRET-C is illustrated using the execution traces in Table 1. We assume the

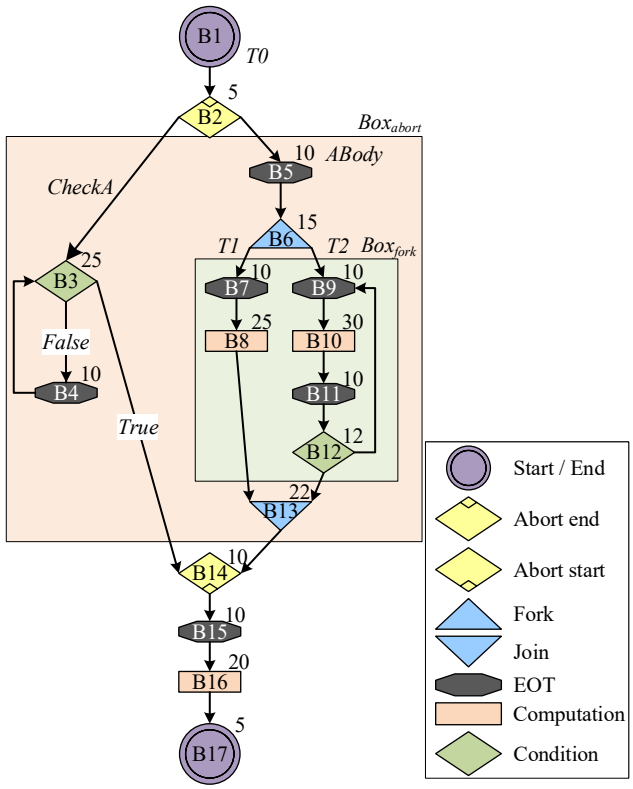


Fig. 5. A TCCFG running example. Used for demonstrating the formulations of WCRT algebra. The TCCFG is divided into boxes based on the hierarchy.

preemption condition is false unless stated otherwise in the event column. Threads in a PRET-C program execute in a static order, from left to right in the TCCFG representation. The execution only switches to the next thread when it reaches an EOT node. The tick count advances when all the active threads have reached their respective EOT nodes.

Table 1. Tick snapshots of the running example.

| Tick count | Execution path | Events tick | Tick cost |
|---|---|---|---|
| 1 | $B1 \rightarrow B2 \rightarrow B3 \rightarrow B4 \rightarrow B5$ | Entering abort | 50 |
| 2 | $B3 \rightarrow B4 \rightarrow B6 \rightarrow B7 \rightarrow B9$ | Forking $T1$ and $T2$ | 70 |
| 3 | $B3 \rightarrow B4 \rightarrow B8 \rightarrow B10 \rightarrow B11$ | $T1$ terminates | 100 |
| 4a | $B3 \rightarrow B4 \rightarrow B12 \rightarrow B13 \rightarrow B14 \rightarrow B15$ | $T2$ terminates, joining | 89 |
| 4b | $B3 \rightarrow B14 \rightarrow B15$ | Preemption | 45 |
| 5 | $B16 \rightarrow B17$ | Program finishes | 25 |

Execution begins from the start node $B1$, and subsequently reaches the abort-start node $B2$. An abort consists of two concurrent threads in a TCCFG: *CheckA* for checking the abort condition and the abort body *ABody*. For a strong abort, *CheckA* has a higher priority than *ABody* and vice versa for a weak abort. These two threads are spawned when the abort start

node executes. The preemption takes place when either of the threads reaches the abort end node *B14*. In our simulation of Table 1, in *tick 1* *CheckA* and *ABody* pause at their EOT nodes *B4* and *B5* respectively. In *tick 2*, the execution resumes from these EOT nodes. The thread *ABody* spawns *T1* and *T2* using the fork node *B6* and suspends itself. In *tick 3*, *T1* terminates as it reaches the join node *B13*, and *T2* pauses at the EOT node *B11*. In *tick 4*, we present two scenarios. If preemption does not take place (*tick 4a*), *T2* terminates and activates the join at *B13* whereupon the EOT node *B15* is reached. However, if preemption takes place (*tick 4b*), *CheckA* reaches the abort-end node *B14*, preempting all threads in the abort body, and the program pauses at *B15*. In either case, the program finishes in *tick 5* by reaching the end node *B17*.

## 4 THE WCRT ALGEBRA

Our WCRT analysis of TCCFGs is based on explicit path enumeration using tick cost automata (TCA) whose timing is captured by formal power series in min-max-plus algebra. We first present the mathematical structure of min-max-plus algebra in Sec. 4.1, then use it to define TCA in Sec. 4.2 and subsequently in Sec. 4.3–4.6 compute the TCA by recursion on the structure of the TCCFG.

### 4.1 Mathematical structure

Min-max-plus algebra $(\mathbb{N}_\infty, \wedge, \oplus, \odot, \mathbb{1}, \mathbb{0})$ is defined over the set $\mathbb{N}_\infty =_{df} \mathbb{N} \cup \{-\infty\}$ of execution times, where $-\infty$ is used to denote inactive transitions or unreachable states. The three operators in the algebra are minimum $\wedge$, maximum $\oplus$, and $\odot$ is addition. All three operators are commutative and associative. The notations $\odot$ and $\oplus$ are chosen to highlight their multiplicative and additive nature. The constants of the algebra are $\mathbb{1} =_{df} 0$ and $\mathbb{0} =_{df} -\infty$. The element $\mathbb{0}$ is absorbant for the operator $\odot$ and neutral for $\oplus$, *e. g.,* $X \odot \mathbb{0} = \mathbb{0}$ and $X \oplus \mathbb{0} = X$. The element $\mathbb{1}$ is neutral for $\odot$, *e. g.,* $X \odot \mathbb{1} = X$. The element $\mathbb{0}$ is also absorbant for the operator $\wedge$, *e. g.,* $X \wedge \mathbb{0} = \mathbb{0}$. The $\wedge$ operator can be used to abstract a number $X \in \mathbb{N}_\infty$ into a boolean $X \wedge \mathbb{1} \in \{\mathbb{0}, \mathbb{1}\}$, *e. g.,* $X \wedge \mathbb{1} = \mathbb{0}$ when $X = \mathbb{0}$ and $X \wedge \mathbb{1} = \mathbb{1}$ when $X \geq \mathbb{1}$. Powers are $X^n = X^{n-1} \odot X$ for $n \in \mathbb{N} \setminus \{0\}$ and $X^0 = \mathbb{1}$.

*4.1.1 Formal power series.* We capture the infinite sequence of ticks in a program as formal power series. A formal power series is a polynomial function with an infinite number of terms, which has a general form of $f(X) = \bigoplus_{i \geq 0} a_i X^i$. For a succinct presentation, the $\odot$ between the coefficient $a_i$ and $X^i$ is often omitted. Each coefficient represents the timing of a tick. We can extract the coefficients from a formal power series using the notation $a_i = [X^i]f(X)$.

A TCCFG does not have instantaneous loops and there is no dynamic creation of threads. Thus, the number of statements executed in any tick, and hence the maximal possible timing cost $[X^i]f(X)$ in any tick $i \geq 0$ is statically bounded by the TCCFG. Moreover, the number of reachable programs states is finite. Hence, the coefficient sequence of $f(X)$ must be ultimately periodic (*e. g.,* Fig. 3). Thus, we can write a formal power series for some $0 \leq n \leq m$ as follows:

$$f(X) = a_0 \oplus a_1 X \oplus \ldots \oplus a_{n-1} X^{n-1} \oplus \bigoplus_{i \geq 0} \left( a_n X^{i(m-n+1)+n} \oplus \ldots \oplus a_m X^{i(m-n+1)+m} \right) \tag{4}$$

$$= a_0 : a_1 : \ldots : a_{n-1} : (a_n : \ldots : a_m)^\omega,$$

where the exponent $\omega$ denotes infinite repetition and the colon : is used as a compact way of writing finite sums of $X^i$-weighted coefficients $a_i$ as a list. The first $n+1$ coefficients occur

only once in the execution. Then the coefficients repeat from $a_{n+1}$ to $a_m$. For instance, the TCCFG $T_\mathrm{A}$ in Fig. 3 generates the coefficient sequence

$$\bigoplus_{i\geq 0}(A1\,X^{3i} \oplus (A2 \oplus A3)\,X^{3i+1} \oplus A4\,X^{3i+2}) = (A1 : (A2 \oplus A3) : A4)^\omega$$

which is an instance of (4) with $n = 0$, $m = 2$ and coefficients $a_0 = A1$, $a_1 = A2 \oplus A3$ and $a_2 = A4$.

The WCRT is the largest coefficient of $f(X)$, which can be computed by setting $X$ to $\mathbb{1}$. This effectively removes all the variable terms $X^n$ since $\mathbb{1}^n = \mathbb{1}$ and leaves the $\oplus$ operator to compute the maximum coefficient:

$$\begin{aligned} f(\mathbb{1}) &= a_0 \oplus (a_1 \odot \mathbb{1}) \oplus \ldots \oplus (a_n \odot \mathbb{1}^n) \oplus ((a_{n+1} \odot \mathbb{1}^{n+1}) \oplus \ldots \oplus (a_m \odot \mathbb{1}^m))^\omega \\ &= a_0 \oplus a_1 \oplus \ldots \oplus a_n \oplus (a_{n+1} \oplus \ldots \oplus a_m)^\omega = \max(a_0, a_1, \ldots a_m). \end{aligned}$$

Instead of $f(\mathbb{1})$ we will also write $f|_{X=\mathbb{1}}$ for the evaluation of the series $f(X)$ at $X = \mathbb{1}$.

*4.1.2 Operations on formal power series.* We will construct the timing equivalent TCA from a TCCFG $G$ hierarchically based on the structure of $G$ by algebraic operations on formal power series. A useful class of operations are the coefficient-wise variants of the binary operators $\wedge$ and $\odot$, denoted $\tilde{\wedge}$ and $\tilde{\odot}$, respectively, and the operator $\oplus$ which is naturally coefficient-wise:

$$\begin{aligned} f_1(X) \oplus f_2(X) &= \bigoplus_{i\geq 0} ([X^i]f_1(X) \oplus [X^i]f_2(X))\,X^i \\ f_1(X)\,\tilde{\wedge}\,f_2(X) &= \bigoplus_{i\geq 0} ([X^i]f_1(X) \wedge [X^i]f_2(X))\,X^i \\ f_1(X)\,\tilde{\odot}\,f_2(X) &= \bigoplus_{i\geq 0} ([X^i]f_1(X) \odot [X^i]f_2(X))\,X^i. \end{aligned}$$

Scalar multiplication with a constant $C$ is given by

$$\begin{aligned} C \odot (f(X)) &= C \odot (a_0 \oplus a_1X \oplus a_2X^2 \oplus \ldots) \\ &= (C \odot a_0) \oplus (C \odot a_1)X \oplus (C \odot a_2)X^2 \oplus \ldots \\ &= \bigoplus_{i\geq 0}(C \odot a_i)X^i \;=\; C^\omega\,\tilde{\odot}\,f(X), \end{aligned}$$

where $C^\omega = \bigoplus_i C\,X^i$ is the constant power series with an infinite repetition of coefficient $C$.

*4.1.3 Shifting the coefficients.* When computing the WCRT, the timing of the current tick may depend on the timing in the previous tick. For example, the outflow of an EOT node depends on whether the EOT node is reachable in the previous tick. We use the $pre(f(X))$ function to produce a one-tick-delay variant of a formal power series $f(X)$. This allows us to access the timing in the previous tick mathematically. The computation of $pre(f(X))$ is as follows:

$$pre(f(X)) = (\mathbb{0} \oplus (f(X) \odot X))\,\tilde{\wedge}\,\mathbb{1}^\omega.$$

In this function, the term $\mathbb{0}$ is added to the series as the $X^0$ coefficient, while the $\tilde{\wedge}$ operator and $\mathbb{1}^\omega$ series reduces the coefficients of the series to be $\mathbb{1}$ or $\mathbb{0}$. For example, $pre(10 : 3 : (\mathbb{0} : 43)^\omega) = \mathbb{0} : \mathbb{1} : \mathbb{1} : (\mathbb{0} : \mathbb{1})^\omega$.

### 4.2 Tick Cost Automata (TCAs)

From a timing stand point, the program execution is a sequence of ticks, and each tick has two possible outcomes. The program can either pause at an EOT node and resume in the next tick, or reach the end node and terminate. TCAs capture timing based on this perspective. A TCA is formed of a sequence of states, and at each state there are two transitions: one leads to the next state denoting the cost to reach the next tick, and one leads to the end state denoting the cost to exit. Since the sequence is ultimately periodic, it will eventually loop back to one of the intermediate state. Fig. 6 shows the general form of a TCA.

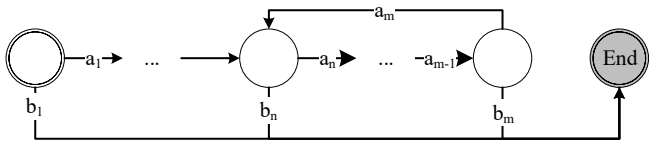


Fig. 6. The general form of a TCA.

Mathematically, we define a TCA $\tau$ as two formal power series: $\tau = (tick, exit)$, where $tick$ captures the cost for reaching the next state (i. e., pausing at an EOT node), and $exit$ captures the cost for reaching the end node. Given Fig. 6, these are:

$$\tau = \begin{cases} tick = & a_1 : a_2 : \ldots : a_{n-1} : (a_n : \ldots : a_m)^\omega \\ exit = & b_1 : b_2 : \ldots : b_{n-1} : (b_n : \ldots : b_m)^\omega. \end{cases}$$

### 4.3 Worst Case Reaction Time Analysis

TCAs are the unit of composition in WCRT algebra. Accordingly, the WCRT of a synchronous program is obtained by constructing the TCA representation of its TCCFG. We compute the WCRT from this TCA. Given a TCA $\tau = (tick, exit)$, the WCRT is

$$\mathrm{WCRT}(\tau) = tick|_{X=\mathbb{1}} \oplus exit|_{X=\mathbb{1}}. \tag{5}$$

The equation (5) takes the maximum of the time cost, over all ticks, to reach a pause (i. e., $tick|_{X=\mathbb{1}}$) or to terminate (i. e., $exit|_{X=\mathbb{1}}$). The computation of the TCA associated with a TCCFG follows the hierarchical structure of the TCCFG. A TCCFG is a single (main) *thread*. A thread is a sequential control flow of nodes and *boxes*. A box consists itself of concurrent threads, thus generating a multi-level hierarchy. For instance, the main thread of the TCCFG in Fig. 5 has nodes *B1*, *B2*, *B14–B17* and the box $Box_{abort}$, which comprises the threads *CheckA* and *ABody*. Thread *ABody* has nodes *B5*, *B6*, *B13* and box $Box_{fork}$. We call $Box_{abort}$ an *abort box* and $Box_{fork}$ a *fork box*.

We define a translation function $TCA(\Phi)$ which traverses this structure by recursive descent and maps each thread or box $\Phi$ into a timing equivalent TCA. The common feature making this possible is that each thread or box has a single start and a unique end point relative to which the timing can be measured out by a TCA. If $G$ is a TCCFG and $T_{main}(G)$ is the main thread, the WCRT of $G$ is

$$WCRT(TCA(T_{main}(G))). \tag{6}$$

In the following subsections, we first define the transformation of a simple thread into a TCA. Then we describe the flattening of *fork* and *abort* boxes.

### 4.4 Modeling the TCA of a thread

A thread can be converted into a TCA only if it has no boxes. If not, the boxes are flattened into a sequence of nodes using the techniques described in Sec. 4.5 and 4.6 later.

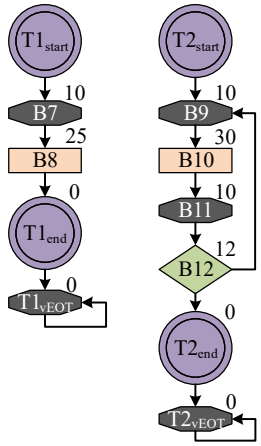


Fig. 7. TCCFG section $Box_{fork}$: concurrent threads *T1* and *T2* with virtual start, virtual end and EOT nodes.

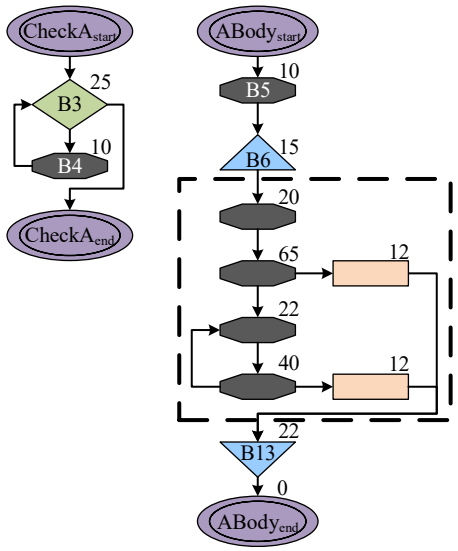


Fig. 8. TCCFG section $Box_{abort}$: the check abort threads *CheckA* and abort body *ABody* with virtual start and virtual end nodes. The box $Box_{fork}$ is replaced with a WCRT equivalent graph of $TCA(Box_{fork})$ indicated by the dashed box

.

When a thread $T$ belongs to a *fork* box or an *abort* box (as opposed to the main thread), it is incomplete in the sense that it is a fragment of the TCCFG with no start and end nodes to mark the beginning and the termination of its execution. To make these threads self-contained,

we add virtual *start* and *end* nodes to them. They are defined as $N_{start}(T) = T_{start}$ and $N_{end}(T) = T_{end}$, respectively, see Fig. 7 and 8.

Additionally, when a thread belongs to a *fork* box, we add a virtual EOT node after its virtual end nodes to mimic the functionality of the join node. A thread terminates when its control flow reaches the end node. However, the joining does not have to take place at the same tick when this happens since sibling threads may still be active. This virtual EOT node serves as a memory of the termination, which is required for formulating the TCA of the box. This is a special EOT node which we denote as $N_{vEOT}(T)$. Those virtual nodes have 0 cost, thus they do not contribute to the WCRT (see Fig. 7).

When threads belong to an *abort* box, virtual EOT are not required since preemption takes place instantaneously. An abort box always has two threads, one for checking the abort condition, and one for the abort body. The execution priority of these two threads depends on the type of the abort: strong or weak. We have developed a formulation that can handle both. In our formulation, we refer to the higher priority thread priority as $T_{high}$ and the lower priority thread as $T_{low}$. *E. g.,* $Box_{abort} = \{T_{high}, T_{low}\}$, where $T_{high} = CheckA$ and $T_{low} = ABody$.

*4.4.1 Start node.* A node is an atomic unit in a TCCFG, and its execution can be captured as a TCA. The start node $N_{start}(T)$ of a thread $T$ (*e. g.,* $N_{start}(T1) = T1_{start}$ in Fig. 7) has the following TCA:

$$tick(N_{start}(T)) = \mathbb{0}^{\omega}$$
$$exit(N_{start}(T)) = \mathbb{1} : \mathbb{0}^{\omega}.$$

This is a generic formulation for a start node. Since the execution of a synchronous program cannot pause at the start node, its formal power series *tick* constantly produces $\mathbb{0}$*s* across all ticks. The start node is the initiator of the program execution. Thus, in the first tick, it produces an execution cost of $\mathbb{1}$ in *exit* (i. e., the execution cost of the program is $\mathbb{1}$ by the time the control flow leaves the start node). Then, the start node remains $\mathbb{0}$ for the rest of the execution, i. e., no control flow leaves the start node.

*4.4.2 Transient nodes.* Transient nodes are *computation*, *condition* and *end* nodes. During execution, transient nodes contribute towards the execution cost when the control flow reaches them, and then they immediately pass on the control flow without pausing (i. e., the control flow exits the transient node in the same tick). Let $precede(N)$ be the set of predecessors of a node $N$ and $C_N$ the associated cost of $N$. The TCA of a transient node $N$ then is:

$$tick(N) = \mathbb{0}^{\omega}$$
$$exit(N) = C_N \odot \left( \bigoplus \{ exit(N_p) \mid N_p \in precede(N) \} \right).$$

The control flow of the program cannot pause at transient nodes, thus the series $tick(N)$ always produces $\mathbb{0}$. The exit cost of a transient node is the maximum cost received from its preceding nodes plus its own cost.

*4.4.3 EOT nodes.* The EOT nodes are the state boundaries of program execution, which take one tick to execute. When the control flow reaches an EOT node, it pauses, and exits

in the next tick. The TCA of an EOT node $N$ is defined as follow:

$$tick(N) = C_N \odot \left(\bigoplus\{exit(N_p) \mid N_p \in precede(N)\}\right)$$
$$exit(N) = pre(tick(N)).$$

The series $tick(N)$ denotes the maximum execution cost to pause at an EOT node, which is equal to the maximum cost from the preceding nodes plus the cost $C_N$ of the EOT node itself. The series $exit(N)$ denotes the exiting of the control flow, which is essentially a one-tick-delayed $tick(N)$ with an execution cost of $\mathbb{1}$. If an EOT node is reachable in a tick, i. e., $[X^n]tick(N) \geq \mathbb{1}$, then its *exit* in the next tick is $[X^{n+1}]exit(N) = \mathbb{1}$, thereby resetting the execution cost to 0 at the beginning of a tick. If an EOT node is not reachable in a tick, i. e., $[X^n]tick(N) = \mathbb{0}$, then the exit will not be produced in the next tick, i. e., $[X^{n+1}]exit(N) = \mathbb{0}$.

*Virtual EOT in fork boxes.* When a thread terminates in a *fork* box, the virtual EOT node is activated by receiving the control flow from the end node, and it remembers the termination through a self-loop. In other words, the virtual EOT nodes are the terminated states of the threads. Let us define $N_{EOT}(T)$ to be the set of EOT nodes of a thread T and $sibling(T)$ the sibling threads of T, that is the thread which share the same box. The TCA of the virtual EOT nodes N of a thread $T$ in a fork box $B$ is defined thus:

$$tick(N) = C_N \odot \left(\bigoplus\{exit(N_p) \mid N_p \in precede(N)\}\right)$$
$$exit(N) = pre(tick(N)) \;\tilde{\odot}\; \left(\bigoplus\{pre(tick(N_s)) \mid N_s \in N_{EOT}(T_s), T_s \in sibling(T)\}\right).$$

The formulation is different from a normal EOT node as we have added a condition in $exit(N)$ to constrain the self-looping behaviour. The self-loop is useful when a thread terminates, while its sibling threads are still in the process of reaching their end nodes and have to pause at an EOT in this tick. The terminated state is remembered for a join in the future. The self-looping should become unreachable (i. e., $\mathbb{0}$) again if none of the sibling threads can pause at an EOT node in the previous tick (i. e., $\bigoplus pre(tick(N_s)) = \mathbb{0}$ in the formulation). The self-loop is active only if, in the previous tick, the virtual EOT node was reachable (i. e., $[X^n]pre(tick(N)) = \mathbb{1}$) and at least one sibling thread could reach an EOT node (i. e., $\bigoplus[X^n]pre(tick(N_s)) = \mathbb{1}$). This prevents generating the TCA state where all the concurrent threads are self-looping at the virtual EOT node and no thread is actually executing.

*EOT in abort boxes.* In the case of an *abort* box $B = \{T_{high}, T_{low}\}$, EOT nodes are modeled differently depending on whether they belong to the $T_{high}$ or $T_{low}$ thread.

- For the EOT nodes $N_i \in N_{EOT}(T_{high})$ in $T_{high}$ we have

$$tick(N_i) = C_{N_i} \odot \left(\bigoplus\{exit(N_p) \mid N_p \in precede(N_i)\}\right)$$
$$exit(N_i) = pre(tick(N_i)) \;\tilde{\odot}\; pre(tick(T_{low})),$$

where we added a condition in *exit* to monitor the status of $T_{low}$. Thread $T_{high}$ resumes its execution from an EOT node only if in the previous tick (1) the EOT node was reachable (i. e., $pre(tick(N_i))$) and (2) $T_{low}$ can reach an EOT node (i. e., $pre(tick(T_{low}))$). If $T_{low}$ cannot reach an EOT node in the previous tick $n$, then $T_{low}$ must have reached the abort-end node, preempting $T_{high}$. Thus, $T_{high}$ must not continue its execution in tick $n + 1$.

- For all EOT nodes $N_i \in N_{EOT}(T_{low})$ in $T_{low} \in B$ we define

$$tick(N_i) = C_{N_i} \odot \left(\bigoplus\{exit(N_p) \mid N_p \in precede(N_i)\}\right)$$
$$exit(N_i) = pre(tick(N_i)) \;\tilde{\odot}\; (tick(T_{high}) \,\tilde{\wedge}\, \mathbb{1}^\omega)$$

and for the start node of $T_{low}$:

$$tick(N_{start}(T_{low})) = \mathbb{0}^\omega$$
$$exit(N_{start}(T_{low})) = (\mathbb{1} : \mathbb{0}^\omega) \;\tilde{\odot}\; (tick(T_{high}) \,\tilde{\wedge}\, \mathbb{1}^\omega).$$

Similar to the formulations for $T_{high}$, we have added a condition in the *exit* of the EOT and start nodes in $T_{low}$. The difference is that, here, we check the status of $T_{high}$ in the current tick. $T_{low}$ can resume/initiate its execution from an EOT/start node only if $T_{high}$ can reach an EOT node in the current tick (i. e., $[X^n](tick(T_{high}) \,\tilde{\wedge}\, \mathbb{1}^\omega) = \mathbb{1}$). If $T_{high}$ cannot reach an EOT node, then it must reach the abort-end node, preempting $T_{low}$ immediately; Thus, $T_{low}$ cannot execute after $T_{high}$ in the same tick.

*4.4.4 TCA of threads.* Finally, the TCA for a thread $T$ is simple. The series $tick(T)$ is the maximum execution cost to reach and pause at an EOT node, and $exit(T)$ is the maximum execution cost to reach the end node and exit:

$$tick(T) = \bigoplus \{tick(N_i) \mid N_\mathrm{i} \in N_{EOT}(T)\}$$
$$exit(T) = exit(N_{end}(T)).$$

### 4.5 Modeling the TCA of a fork box

When the threads making a *fork* box $B$ are all reduced as TCAs, we can then use them to express the TCA of the *fork* box itself. This captures forking, concurrent execution and the joining of threads:

$$tick(B) = \tilde{\bigodot}\{tick(T_i) \oplus tick(N_{vEOT}(T_i)) \mid T_i \in B\}$$
$$exit(B) = \left(\tilde{\bigodot}\{tick(N_{vEOT}(T_i)) \mid T_i \in B\}\right) \;\tilde{\odot}\; \left(\mathbb{1}^\omega \;\tilde{\wedge}\; \bigoplus\{exit(T_i) \mid T_i \in B\}\right).$$

The series $tick(B)$ denotes the timing of concurrent execution, where the control flow pauses in the box. The coefficients are computed by summing up the maximum execution time of all the threads. Here, the virtual EOT nodes are included in the calculation, hence the maximum execution time of a thread is $tick(T_i) \oplus tick(N_{vEOT}(T_i))$. This is because if the control flow pauses in the box (i. e., no joining), reaching an end node is equivalent to pausing at an EOT node (*e. g., tick 3* in Table. 1).

The series $exit(B)$ denotes the timing of joining, where the control flows from the concurrent threads merge and exit the box. This formulation has two parts. The first part is the computation of the cost $\tilde{\bigodot} tick(N_{vEOT}(T_i))$. A joining takes place when all the threads have terminated, hence the joining cost is computed by summing up the costs for reaching the virtual EOT nodes (i. e., terminated states). If any thread is not yet terminated, that is, $[X^n]tick(N_{vEOT}(T_i)) = \mathbb{0}$, the joining cost $[X^n]exit(B)$ is $\mathbb{0}$. The second part of the formulation is a condition for joining: $\mathbb{1}^\omega \,\tilde{\wedge}\, \bigoplus exit(T_i)$. A joining takes place when the last child thread terminates, where at least one thread should be able to exit in that tick ($[X^n]exit(T_i) \geq \mathbb{1}$). The term $\bigoplus exit(T_i)$ is $\mathbb{0}$ if no thread can exit. This condition prevents a join from occurring when all the virtual EOT nodes are self-looping but no thread

can reach the end node. This second part is a boolean series, where the $\tilde{\wedge}$ operator and $\mathbb{1}^\omega$ reduce the coefficients to $\mathbb{1}$ or $\mathbb{0}$. Therefore, the second part does not contribute to the WCRT. Finally, we compose the two parts using the $\tilde{\odot}$ operator so that $[X^n]exit(B) = \mathbb{0}$ if either part is $\mathbb{0}$.

As an example, here is a part of the equation system that describes the TCA of $Box_{fork}$ in Fig. 7:

$$
\begin{aligned}
tick(B12) &= \mathbb{0}^\omega \\
exit(B12) &= C_{B12} \odot exit(B11) \\
tick(B9) &= C_{B9} \odot (exit(T2_{start}) \oplus exit(B12)) \\
exit(B9) &= pre(tick(B9)) \\
tick(T2_{EOT}) &= exit(T2_{end}) \oplus exit(T2_{EOT}) \\
exit(T2_{EOT}) &= pre(tick(T2_{EOT})) \,\tilde{\odot}\, pre(tick(B7)) \\
tick(T2) &= tick(B9) \oplus tick(B11) \\
exit(T2) &= exit(T2_{end}) \\
tick(Box_{fork}) &= (tick(T1) \oplus tick(T1_{EOT})) \,\tilde{\odot}\, (tick(T2) \oplus tick(T2_{EOT})) \\
exit(Box_{fork}) &= tick(T1_{EOT}) \,\tilde{\odot}\, tick(T2_{EOT}) \,\tilde{\odot}\, (\mathbb{1}^\omega \,\tilde{\wedge}\, (exit(T1) \oplus exit(T2))).
\end{aligned}
$$

### 4.6 Modeling the TCA of an abort box

The TCA of an abort box $B$ is defined as follows:

$$
\begin{aligned}
tick(B) &= tick(T_{high}) \,\tilde{\odot}\, tick(T_{low}) \\
exit(B) &= exit(T_{high}) \oplus (tick(T_{high}) \,\tilde{\odot}\, exit(T_{low})).
\end{aligned}
$$

The series $tick(B)$ captures the timing when no preemption occurs. It sums up the execution time of $T_{high}$ and $T_{low}$ as they execute concurrently. If either thread terminates in a tick (i. e., $[X^n]tick(T_{high}) = \mathbb{0}$ or $[X^n]tick(T_{low}) = \mathbb{0}$), this triggers a preemption and $[X^n]tick(B) = \mathbb{0}$. The series $exit(B)$ obtains the timing when preemption occurs. $T_{high}$ and $T_{low}$ trigger a preemption differently. When $T_{high}$ triggers a preemption, the exit cost of the box is the exit cost of $T_{high}$ (i. e., $exit(T_{high})$), since $T_{low}$ is preempted. However, when $T_{low}$ triggers a preemption, it implies $T_{high}$ had executed and reached an EOT node, as $T_{high}$ has a higher execution priority. Thus, the exit cost in this case is $tick(T_{high}) \,\tilde{\odot}\, exit(T_{low})$. Overall, the exit cost of the box is the maximum of the two cases.

As an example, here is a part of the equation system that describes the TCA of $Box_{abort}$. Note that the WCRT analysis has first flattened the *ABody* thread by substituting the box

$Box_{fork}$ with a WCRT equivalent graph derived from its TCA.

$$
\begin{aligned}
tick(B4) &= C_{B4} \odot exit(B3) \\
exit(B4) &= pre(tick(B4)) \;\tilde{\odot}\; pre(tick(ABody)) \\
\\
tick(B5) &= C_{B5} \odot exit(ABody_{start}) \\
exit(B5) &= pre(tick(B5)) \;\tilde{\odot}\; (\mathbb{1}^{\omega} \,\tilde{\wedge}\, tick(CheckA)) \\
\\
tick(ABody_{start}) &= \mathbb{0}^{\omega} \\
exit(ABody_{start}) &= (\mathbb{1} : \mathbb{0}^{\omega}) \;\tilde{\odot}\; (\mathbb{1}^{\omega} \,\tilde{\wedge}\, tick(CheckA)) \\
\\
tick(CheckA) &= tick(B4) \\
exit(CheckA) &= exit(CheckA_{end}) \\
tick(Box_{abort}) &= tick(CheckA) \;\tilde{\odot}\; tick(ABody) \\
exit(Box_{abort}) &= exit(CheckA) \oplus (tick(CheckA) \,\tilde{\odot}\, exit(ABody)).
\end{aligned}
$$

## 5 IMPLEMENTATION

We have implemented our timing simulation using Python. The core is to flatten a TCCFG box to TCA. Given a box as input, we first generate the equation system that describes the control flow of the box with respect to timing. The TCA of the box is generated by repeatedly expanding the equations. Each iteration of expansion produces one of the TCA states, and the analysis finishes when the TCA loops back to an explored state, or reaches the end state. The worst-case complexity of this algorithm is exponential with the size of the TCCFG. However, as our evaluation shows, on typical TCCFGs from PRET-C programs, this worst-case does not occur.

## 6 EVALUATION

In this section, we present an experimental evaluation of WCRT algebra through benchmarking, and compare it with the state-of-the-art model checking based [4] and ILP based techniques [16]. The benchmarking was conducted in two phases. The first phase evaluates performance using a set of PRET-C programs taken from [4, 17, 18]. The second phase examines the theoretical properties of WCRT algebra using a set of synthetic TCCFGs.

### 6.1 Benchmark settings

The benchmarking was carried out on a Windows based computer, with a i5-6300U processor and has 8GB of RAM. Both of the model checking based and ILP based techniques used pre-existing tool implementations that could be adapted to TCCFG without modification to the algorithms. The benchmarking process is to apply all three techniques over the same set of TCCFGs, and record the computed WCRTs and their analysis times. In the sequel, we refer to our timing simulation in WCRT algebra as 'WA', to the model checking approach as 'MC' and to the ILP approach as '$\text{ILP}_{\text{n}}$'.

*6.1.1 Model checking based approach.* The model checking based approach (MC) is based on UPPAAL [2]. The technique first transforms the TCCFG into a functionally equivalent UPPAAL model, with an additional variable to track the execution time. This variable is incremented each time a transition takes place and is reset when advancing to the next tick. The WCRT is computed through a query on the maximum value of the variable.

MC is a more capable technique than the other two in the sense that it can optionally track variable values at the expense of a longer analysis time. For a meaningful comparison, in this benchmarking, we configure the model checking approach to analyse for tick alignment only.

*6.1.2 ILP based approach.* The ILP based approach ($ILP_n$) [16] uses iterative narrowing to tackle the tick alignment problem without sacrificing scalability. An ILP model is used as the baseline, and its outcome is verified for tick alignment in each iteration using an ILP based verification technique. If the verification fails, the baseline model is refined using the verification results, and the analysis continue to the next iteration. This continues until the verification is passed. In $ILP_n$, the Gurobi Optimiser [1] is used for solving ILP.

## 6.2 Existing benchmark programs

The details of the benchmark programs and the analysis results from the three techniques are summarised in Table 2. The benchmark programs cover a range from small to large problem sizes. For example, the largest program is *WaterMonitor* which has 3204 lines of C code generated from the PRET-C specification consisting of 40 threads. All the techniques produce the same WCRT estimate, which cross-checks the correctness of all three techniques.

Table 2. Benchmarking results for model checking (*MC*), $ILP_n$ and WCRT algebra (*WA*).

| Name | LOC | Thd | Analysis Time (s) | | | WCRT |
|---|---|---|---|---|---|---|
| | | | MC | $ILP_n$ | WA (states) | |
| ChannelProtocol | 591 | 7 | 2.63 | 0.10 | 0.13 (25) | 997 |
| Flasher | 816 | 7 | 3.22 | 0.11 | 0.28 (23) | 617 |
| RobotSonar | 962 | 7 | 6.51 | 0.63 | 0.48 (23) | 1874 |
| Synthetic 1 | 1287 | 7 | 12.00 | 4.55 | 0.51 (21) | 2218 |
| Synthetic 2 | 1293 | 7 | 12.6 | 2.10 | 0.56 (21) | 2514 |
| DrillStation | 1094 | 15 | 5.30 | 2.80 | 0.18 (49) | 2751 |
| CruiseControl | 2302 | 25 | N/A | 0.72 | 0.43 (29) | 1931 |
| RailroadCrossing | 2713 | 30 | N/A | 1.05 | 0.57 (37) | 4472 |
| WaterMonitor | 3204 | 40 | N/A | 0.65 | 0.50 (24) | 4631 |

The analysis time of MC increases exponentially as the number of program states increases. Eventually, it is not able to finish the analysis for the three largest programs after two minutes due to insufficient memory. This is a typical behaviour for explicit path enumeration, which is the reason that the approach is deemed to be unsuitable for timing analysis. In comparison, $ILP_n$, which is based on implicit path enumeration, is much more scalable. On average, $ILP_n$ only takes 1.41 seconds, with a peak of 4.55 seconds.

However, WA is even faster than $ILP_n$, which breaks the conventional belief about timing analysis: explicit path enumeration does not scale well. The average analysis time of WA is 0.4 second, which is 3.5 times faster than $ILP_n$. Moreover, the current implementation of WA involves a disk access delay as it writes programs to disk, and then reads them to execute. If we exclude the disk access delay, the actual analysis time of WA can be considered instantaneous.

The prime factor for the short analysis time is the number of TCA states generated. The total number of accumulated TCA states, that is the sum of all TCA states for boxes and the main thread, is shown in parenthesis after the analysis time of WA. We observe

that these accumulated TCA states are very small, some of them being even less than the number of threads in the program (i. e., less than one state per thread). This indicates that for realistic synchronous program structure the state explosion problem does not occur and that WA is able to benefit from this well-behaved structure. On the other hand, MC, which generated millions of states, though working at the same abstraction level, runs into a (algorithm-induced) combinatorial explosion.

### 6.3 Theoretical properties

WCRT algebra has demonstrated exceptional performance on our benchmark programs, and we believe this is because the state explosion is mitigated by the synchronous composition of TCAs.

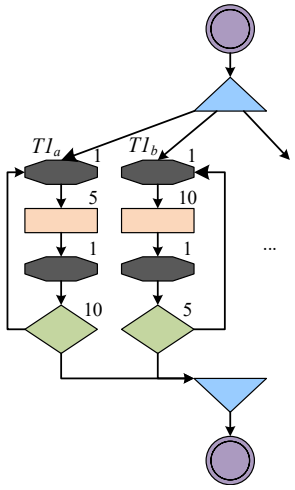


Fig. 9. Synthetic A.

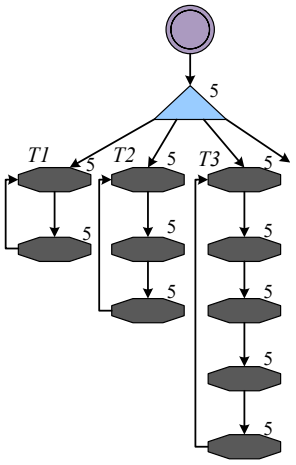


Fig. 10. Synthetic B.

To evaluate this hypothesis, we benchmarked the techniques using a set of highly symmetrical TCCFGs shown in Fig. 9. The number of threads is incremented by replicating $T1_a$ and $T1_b$ alternatively. This set of synthetic TCCFGs is taken from [16] which was originally designed to stress $\mathrm{ILP_n}$.

The results for the first set of synthetic benchmarks are shown in Fig. 11. As expected, WCRT algebra only has to generate three TCA state to compute the WCRT regardless of number of threads. Thus, it results in an analysis time constantly below 0.1 second. In contrast, MC exhibits the same exponential trend as before, and $\mathrm{ILP_n}$ also increases exponentially in this case.

On the other hand, how can we see combinatorial explosion in our timing simulation in WCRT algebra? This happens in cases where the parallel TCAs are highly decoupled in terms of tick alignment. The corner case that triggers the worst-case scenario for WA is shown in Fig. 10. The synthetic TCCFGs have an increasing number of threads, and each thread is a sequence of EOT nodes executing in a loop. The key here is the number of EOT nodes, which are prime numbers, starting from 2, then 3, 5 and so on. The total TCA states generated by simulation is the product of the prime numbers, which is the same as the state-space considered by the MC approach.

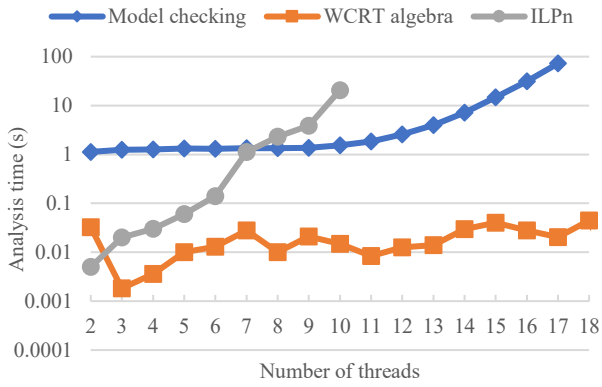


Fig. 11. Results of the model checking, $ILP_n$ and timing simulation in WCRT algebra with Synthetic A.

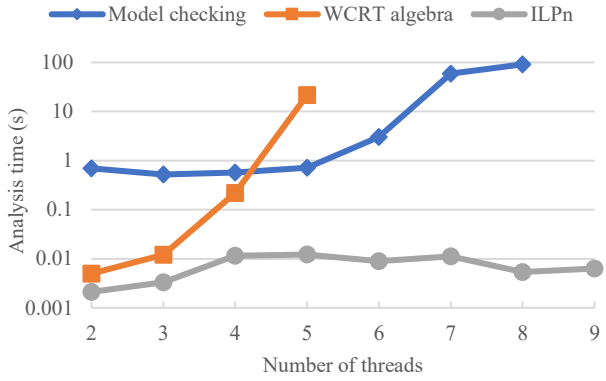


Fig. 12. Results of the model checking, $ILP_n$ and timing simulation in WCRT algebra with Synthetic B.

The results of this second phase is shown in Fig. 12. The analysis times of WA and MC grow exponentially as the number of threads (i. e., program states) increases. However, WA is significantly worse than MC. WA is only able to analyse up to five threads, while MC can analyse up to eight threads. WA may seem faster for the first four benchmarks, but the differences are negligible. This difference in performance is a result of the implementation. The Python implementation of WA is much slower than the native binary of the UPPAAL model checker used in MC. Also, we believe the Python implementation is less optimised. On the other hand, $ILP_n$ exhibits the typical behaviour of implicit path numeration, which scales well with respect to program states.

## 7 CONCLUSIONS

Conventional experience is that explicit path enumeration gives precise results but scales poorly as the number of concurrent threads increases. Thus, it is rarely used for WCRT analysis of synchronous programs. The problem is that the existing approach tries to compute

the WCRT, a non-functional property, based on *functionally* equivalent automata. In this paper, we presented a new analysis via pairs of formal power series in min-max-plus algebra, called TCAs, which is based on the idea of *timing* equivalence. We simplify threads as WCRT equivalent TCAs before composing them, which greatly reduces the search space. We could show empirically that WCRT algebra, on the given benchmark examples, is much faster than the most widely used existing approaches.

Currently, our work is restricted to TCCFG execution structures obtained from PRET-C source programs. We plan to extend the algorithm to cover signals and additional types of preemption operators in order fully to model languages like Esterel, SCCharts or Safe State Machines. At the same time, there is potential to improve both the efficiency and the precision of our analysis.

Regarding efficiency, we plan to use specialised algebraic transformations to deal with "hard" programs like Synthetic B. Specifically, we conjecture that the tick expressions in the $\text{ILP}_n$ approach [16] can be solved analytically in our algebraic approach. In [3] we sketch a technique based on tick expression (an abstract form of frequency domain modeling) that reduces the computation of WCRT for parallel compositions to the maximum weighted clique problem for so-called *tick alignment graphs*. We expect this to help us recover in WCRT algebra some of the efficiency of the $\text{ILP}_n$ method exhibited in Fig. 12. Also, the Python implementation leaves ample room for optimisations.

Regarding precision, the main limitation is the lack of data-dependency in our modelling. It is known that signals and boolean data an be expressed in min-max-plus algebra, too, using the negation operator [12]. Exploiting this, we plan to extend our approach to permit tracking of data/signal variables to capture state invariants like in [13].

## REFERENCES

[1] 2016. Gurobi optimiser. (Nov. 2016). http://www.gurobi.com
[2] 2016. UPPAAL model checker. (Nov. 2016). http://www.uppaal.org
[3] J. Aguado, M. Mendler, J.J. Wang, B. Bodin, and P. Roop. 2017. Compositional Timing-Aware Semantics for Synchronous Programming. In *Forum on Specification and Design Languages (FDL'2017)*. Verona, Italy.
[4] Sidharta Andalam, Partha S. Roop, and Alain Girault. 2011. Pruning infeasible paths for tight WCRT analysis of synchronous programs. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–6.
[5] Sidharta Andalam, Partha S. Roop, Alain Girault, and Claus Traulsen. 2014. A Predictable Framework for Safety-Critical Embedded Systems. *IEEE Trans. Comput.* 63, 7 (July 2014), 1600–1612.
[6] Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden. 2008. Compilation and worst-case reaction time analysis for multithreaded Esterel processing. *EURASIP Journal on Embedded Systems* (2008), 594129.
[7] Reinhold Heckmann and Christian Ferdinand. 2005. Verifying safety-critical timing and memory-usage properties of embedded software by abstract interpretation. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*. 618–619.
[8] Lei Ju, Bach Khoa Huynh, Samarjit Chakraborty, and Abhik Roychoudhury. 2009. Context-sensitive timing analysis of Esterel programs. In *Proceedings of the Design Automation Conference (DAC)*. 870–873.
[9] Lei Ju, Bach Khoa Huynh, Abhik Roychoudhury, and Samarjit Chakraborty. 2012. Performance Debugging of Esterel Specifications. *Real-Time System* 48, 5 (Sept. 2012), 570–600.
[10] Matthew Kuo, Roopak Sinha, and Partha S. Roop. 2011. Efficient WCRT analysis of synchronous programs using reachability. In *Proceedings of the Design Automation Conference (DAC)*. 480–485.
[11] Yau-Tsun Steven Li and Sharad Malik. 1995. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of Languages, Compilers and Tools for Real-time Systems (LCTES)*, Vol. 30. ACM, 88–98.

[12] Michael Mendler, Partha S. Roop, and Bruno Bodin. 2016. A Novel WCET Semantics of Synchronous Programs. In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*. Springer, 195–210.
[13] Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Fabienne Carrier, and Mihail Asavoae. 2015. Timing analysis enhancement for synchronous program. *Real-Time Systems* 51, 2 (2015), 192–220.
[14] Partha S. Roop, Sidharta Andalam, Reinhard von Hanxleden, Simon Yuan, and Claus Traulsen. 2009. Tight WCRT analysis of synchronous C programs. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 205–214.
[15] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. 2014. SCCharts: Sequentially Constructive Statecharts for Safety-critical Applications: HW/SW-synthesis for a Conservative Extension of Synchronous Statecharts. In *Programming Language Design and Implementation (PLDI)*. ACM, 372–383.
[16] Jia Jie Wang, Partha S. Roop, and Sidharta Andalam. 2013. ILPc: A novel approach for scalable timing analysis of synchronous programs. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 1–10.
[17] Li Hsien Yoong, Partha S. Roop, and Zoran Salcic. 2013. Implementing constrained cyber-physical systems with IEC 61499. In *ACM Transactions on Embedded Computing Systems (TECS)*. Number 1. ACM.
[18] Li Hsien Yoong and Gareth D. Shaw. 2010. Auckland Function Block Benchmark. University of Auckland. (2010). pretzel.ece.auckland.ac.nz/files/iec61499-benchmarks.zip

Received April 2017; revised May 2017; accepted June 2017