# Context-Sensitive Timing Analysis of Esterel Programs

Lei Ju Bach Khoa Huynh Samarjit Chakraborty Abhik Roychoudhury
Department of Computer Science, National University of Singapore
{julei, huynhbac, samarjit, abhik}@comp.nus.edu.sg

## ABSTRACT

Traditionally, synchronous languages, such as Esterel, have been compiled into hardware, where timing analysis is relatively easy. When compiled into software – e.g., into sequential C code – very conservative estimation techniques have been used, where the focus has only been on obtaining safe timing estimates and not on the cost of the implementation. While this was acceptable in avionics, efficient implementations and hence tight timing estimates are needed in more cost-sensitive application domains. Lately, a number of advances in Worst-Case Execution Time (WCET) analysis techniques, coupled with the growing use of software in domains such as automotives, have led to a considerable interest in timing analysis of code generated from Esterel specifications. In this paper we propose techniques to obtain tight estimates on the processing time of input events by sequential C code generated from Esterel programs. Execution of an Esterel program – as in all other synchronous languages – is logically made up of a sequence of *clock ticks*. In reality, they take non-zero time which depends on the generated C code as well as the underlying hardware platform on which this code is executed. Apart from exploiting the specific structure of this C code to obtain tight WCET estimates, we capture program-level *contexts* across ticks in order to obtain tight estimates on response times of events whose processing spans across multiple clock ticks. Such tighter estimates immediately translate into more cost-effective implementations. Our experimental results with realistic case studies show 30% reduction in timing estimates when program level context information is taken into account.

## Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Real-time and embedded systems

## General Terms

Design, Languages, Performance

## Keywords

Esterel, Synchronous programming, Worst-case Execution Time (WCET) analysis

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 2009,* July 26 - 31, 2009, San Francisco, California, USA.
Copyright 2009 ACM ACM 978-1-60558-497-3 -6/08/0006 ...$10.00.

## 1. INTRODUCTION

Generating implementations from synchronous language specifications is widely practiced in safety-critical domains such as avionics where certification of the generated implementation is essential. Programs in synchronous languages are considered to execute as a sequence of *clock ticks*. The processing of all events arriving within a single tick are assumed to happen instantaneously or in zero time, which is referred to as the *perfect synchrony hypothesis*. This logical view of program execution offers a clean semantics and greatly simplifies the programming and verification of concurrent reactive systems. An implementation generated from a synchronous language specification is said to follow the synchrony hypothesis if all events that are logically assumed to be processed instantaneously are processed before the next set of events arrive. Hence, for the synchrony hypothesis to hold, the estimated Worst-case Execution Time (WCET) associated with the processing of events should be less than the minimum separation time between the arrival of sets of events (that are assumed to be processed instantaneously).

Verifying the synchrony hypothesis when a synchronous language program is compiled into hardware is relatively straightforward. When such languages are compiled into software – e.g., into a sequential C code – the associated WCET analysis is more complicated and depends both on the generated code, as well as on the micro-architecture of the platform executing this code. For an arbitrary C program running on a modern processor, determining its execution time accurately is an intractable problem and also requires a substantial amount of user intervention. As a result, when performing timing analysis of software generated from synchronous languages, it is difficult to produce very tight timing estimates. This is acceptable in domains such as avionics where the only focus is on safety and formal certification.

However, with the growing use of software in domains such as automotives – which are significantly more cost-sensitive – there is an increasing need to develop timing analysis techniques that return *safe* as well as *tight* timing estimates. Tighter timing estimates clearly lead to more cost-effective, resource-saving designs. There has also been a renewed interest in timing analysis of software generated from synchronous languages, because of the recent advances in WCET analysis techniques and the availability of industry-strength tools (e.g., [1, 9]). Very recently, plans to integrate the synchronous language SCADE from Esterel Technologies with the aiT WCET analyzer from AbsInt GmbH [1], targeting general-purpose processors, was reported in [7].

**Our contributions:** In this paper we propose techniques for tight response time analysis of sequential C code generated from the popular synchronous language Esterel [3]. Our techniques are fairly general and would extend to other synchronous languages as well (such as Lustre/SCADE) with minor modifications. Note that the

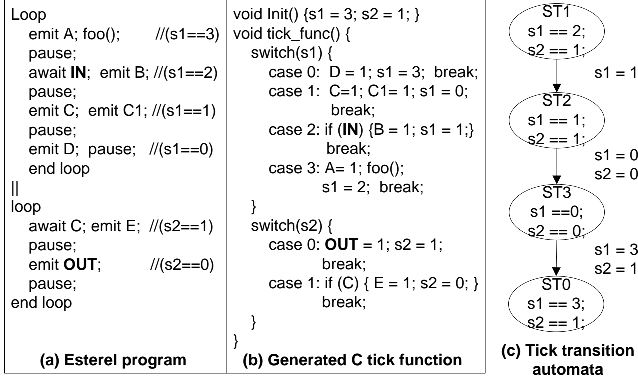


**Figure 1: An Esterel program, compiled C tick function, and Tick Transition Automata**

central problem here is to verify whether the synchrony hypothesis holds for a particular C code and underlying processor architecture. Towards this, we systematically remove infeasible program paths in the generated C code by exploiting the syntax and semantics of the source Esterel program. Further, we capture program contexts at the start of any clock tick in order to further refine our execution time estimates. Program contexts might comprise values of certain variables at the start of a clock tick, which might help in ruling out certain execution paths in the code to be executed within this tick. Capturing such program contexts lead to significantly improved response time estimates of events whose processing spans multiple clock ticks.

**Related work:** Analysis and debugging of timing properties of synchronous language specifications (targeting general-purpose processors) has been somewhat ignored until recently. High-level timing analysis of Esterel programs have been studied in [8, 15], where the problem was to compute the number of transitions in the underlying automata (encoding an Esterel specification) in response to different input events. In other words, the problem is that of computing the number of Esterel clock ticks, rather than the execution time of code within a tick. The timing analysis problem where the states of the automata have been annotated with WCET estimates has been discussed in [11].

Low level WCET analysis for a single Esterel tick is solved in [2] for a special Esterel processor, where the instruction set and micro-architecture are different from a general-purpose processor. [14] has been addressed the problem of infeasible paths in the generated code is mentioned and timing analysis of the whole Esterel program is studied. The methodology is restricted, since it requires two separate codes to be generated from the synchronous program — one on which the WCET analysis is performed, and one which guides the analysis. Again, the issue of capturing context information has not been addressed.

Finally, very recently, [10] and [7] report preliminary results on integrating stand-alone WCET analyzers with synchronous language compilers to provide real-time guarantees on the generated C code. The framework we propose here follows a similar direction, and we believe that our techniques can be integrated with the works in [10] and [7] to obtain even tighter timing estimates.

## 2. OVERVIEW OF ESTEREL

Figure 1(a) shows a toy Esterel program which will be used as an illustrative example in this paper. It consists of two concurrently running processes. The input event $IN$ is consumed by the first process in the second logical tick, and the corresponding output event $OUT$ is emitted by the second process in the fourth tick. Thus, it takes three logical ticks to produce the output. The first tick in the first process performs some other operations (e.g., system initialization) that are irrelevant to the computation of $IN$. Details of the syntax and semantics of Esterel may be found in [3, 6].

**Compiling Esterel:** In our problem setting, we assume an Esterel program to be compiled into C code and executed on a single-core general purpose processor. Various techniques exist for compiling Esterel into C programs [13]. In this paper, we focus our discussion on the control flow graph-based Esterel compilation, which normally produce fast and small C code. Figure 1(b) shows (a simplified version of) the generated C code from the example Esterel program using the open source Columbia Esterel Compiler (CEC) [5]. The compiled C code is in the form of a tick function, such that the set of (feasible) execution paths of the tick function represent all possible computations in an Esterel tick.

## 3. TICK TRANSITION AUTOMATA

Esterel language is finite state in nature, that is, a finite-state automata can capture the behavior of an Esterel program. The full automata corresponding to an Esterel program has many uses, such as in compilation and/or program property verification. However, the combinatorial explosion in the number of states of the full automata is well-known. Instead, for our response-time calculation, we construct a smaller automata called the Tick Transition Automata (TTA for convenience, but not to be confused with Time triggered architectures). The states of this automata capture only the global control flow of an Esterel program — data variable values do not appear in the states.

As already mentioned above, compiling an Esterel program generates a tick function which captures all possible Esterel executions in a single tick. In other words, the Esterel program execution corresponds to repeated executions of this tick function. Naturally, the tick function needs auxiliary variables to capture the progress in control flow in each process (or thread) of the Esterel program. For each process $i$, a state variable $s_i$ is introduced. Different values of a state variable $s_i$ correspond to the different ticks that process $i$ could execute.

**Control states:** The states of the TTA correspond to valuations of the $s_i$ variables. A transition in TTA corresponds to assignment of one or more $s_i$ variables. In the example shown in Figure 1, two state variables $s1$ and $s2$ are introduced to encode the tick transition information of the Esterel program. The states of the TTA correspond to the valuations of $s1, s2$. We call a valuation of the $s_i$ variables as a *global control state* since it captures the progress in control flow of all the threads of an Esterel program. The individual $s_i$ variables will be called as *control state variables*.

**Formal definitions:** Formally, a TTA identifies all paths in the Esterel program that can be executed between an input event *IN* and its output *OUT*. It can be defined as a 5-tuple $\langle Q, \Sigma, \delta, Q_0, F \rangle$, where

- $Q$ is the set of all TTA states. A TTA state is a global control state capturing the progress in control flow in all the program threads.
- $\Sigma$ is a finite set of symbols, where each symbol represents a value assignment on one or more state variables.
- $\delta$ is the transition function, such that $\delta : Q \times \Sigma \rightarrow Q$. Each transition in the automata represents an execution of a tick in the Esterel program.
- $Q_0, F$ are the set of initial/final states of the TTA. An initial state is a global control state of the Esterel program where the input signal *IN* is ready to be consumed. A final state is one where the output signal *OUT* is produced.

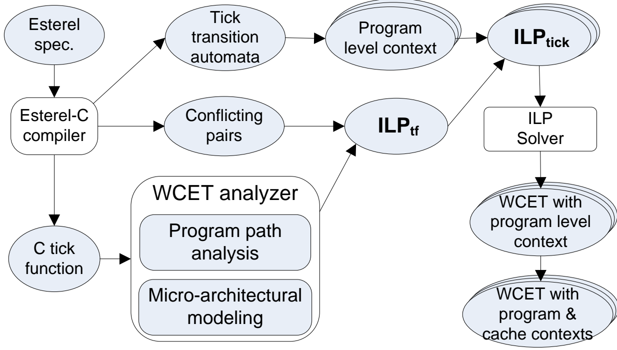

**Figure 2: Framework to compute WCET for a single tick**

Figure 1(c) shows the TTA between the input event *IN* and its output $OUT$ for the example Esterel program and generated C tick function. In between the initial state $ST1$ and final state $ST0$, there is only one possible execution path which consists of three ticks.

# 4. WCET ESTIMATION

We first describe how to compute WCET of a single Esterel tick in the compiled C program. Then we show how to incorporate program contexts captured by TTA into the analysis to obtain tighter WCET estimate. An overview of the analysis framework is shown in Figure 2.

## 4.1 WCET of a Single Tick

Since the time required to produce a designated output *OUT* in response to a given input *IN* may involve several ticks, the first step is to tightly bound the Worst-case Execution Time (WCET) of a single tick.

The WCET of any single Esterel tick can be calculated using a static analysis based WCET estimation on the generated C tick function. The generated C code normally contains huge number of infeasible paths that do not correspond to any concrete program execution traces. In order to obtain a tighter estimate, specifically designed infeasible path analysis can be used to effectively exclude infeasible paths from being considered as the WCET path. Readers are referred to [10] for a detailed description of the single tick WCET analysis. Note that such an analysis is context-insensitive, due to the lack of consideration of program states before execution of a particular Esterel tick. In this following discussion, let us refer to this estimate as $WCET_{tf}$.

# 5. HANDLING PROGRAM CONTEXTS

So far, we have elaborated on the WCET estimation of *any* given tick. Our goal now is to study whether our estimation method is *context-aware*. In other words, we want to estimate the WCET of a tick $ST_i \rightarrow ST_j$ (where $ST_i$ and $ST_j$ are states of the TTA) by considering the program states in which the tick is executed. The program-level context with which $ST_i \rightarrow ST_j$ is executed is captured by $ST_i$. We describe how such context information is integrated into our WCET estimation for a single tick.

As mentioned in the previous section, we assume that each tick between input *IN* and output *OUT* takes time $WCET_{tf}$ (obtained by solving $ILP_{tf}$, see Figure 2). However, this clearly leads to a gross over-estimation. To accurately estimate the worst-case response time between *IN* and *OUT*, our first step is to accurately estimate the WCET of each individual tick between *IN* and *OUT*. Thus, we accurately estimate the WCET of each transition in the TTA. This is achieved by generating additional constraints for each specific transition, and integrating them with the tick function's WCET constraints $ILP_{tf}$ to build a new ILP formulation. This yields an ILP formulation $ILP_{tick}$ for each tick in the TTA (see Figure 2). Solving $ILP_{tick}$ will produce the accurate WCET estimate of the specific tick in question.

We now explain how the additional ILP constraints for a specific tick transition $ST_i \rightarrow ST_j$ are generated. The key difficulty here is that the ILP constraints refer to occurrences of code fragments in the generated C code – they *do not* refer to occurrences of specific ticks at the Esterel program level. Hence, constraints resulting from the occurrence of a specific tick $ST_i \rightarrow ST_j$ need to be expressed in terms of *occurrences of nodes in the Sequential Control flow graph (SCFG) of the code generated from Esterel.*

Recall that $ST_i$ and $ST_j$ correspond to valuations of control state variables $s_1, \ldots, s_n$ where $n$ is the number of threads in the Esterel program. Now, let the value of a state variable $s_k$ in $ST_i$ be $v$, and suppose it is assigned to a new value $v'$ in the tick transition $ST_i \rightarrow ST_j$. For each edge $B \rightarrow B'$ from node $B$ to $B'$ in the sequential Control Flow Graph (SCFG) of the generated C code, if $B$ contains a test on $s_k$ and the edge can be taken only when "$s_k! = v$", the following ILP constraint is generated: $\{E_{B \rightarrow B'} = 0\}$. Here, $E_{B \rightarrow B'}$ is the number of times control flows through the SCFG edge $B \rightarrow B'$. These path constraints ensure that the tick execution that corresponds to $ST_i \rightarrow ST_j$ takes only the SCFG path where "$s_k == v$" whenever state variable $s_k$ is tested, for each state variable $s_k$ in the TTA state $ST_i$.

Moreover, there can be multiple outgoing tick transitions from a TTA state $ST_i$. Suppose the control state variable $s_k$ is assigned to a new value $v'$ ("$v! = v'$") in the tick transition $ST_i \rightarrow ST_j$. To calculate the WCET for a particular tick from $ST_i \rightarrow ST_j$, we simply add ILP constraints to ensure that state variable $s_k$ is assigned to $v'$ during the tick execution. Let $\mathcal{B}$ be the set of SCFG nodes that contain the assignment "$s_k = v'$". We set $\{\sum_{B \in \mathcal{B}} N_B > 0\}$ where $N_B$ is the execution count of a node $B \in \mathcal{B}$.

# 6. WCRT ESTIMATION

Our TTA captures all execution paths between the consumption of a given input *IN* and the production of an output *OUT*. Given a TTA and tight WCET values for tick transitions in it, we now need to compute the worst case response time (WCRT) between an input signal *IN* and output signal *OUT*.

Since the execution count of each tick transition is an integer, we employ an Integer Linear Programming (ILP) approach to compute the WCRT. We solve the following ILP optimization problem. This problem uses the WCET values of the individual ticks as constants.

$$\text{maximize} \sum_{ST_i \rightarrow ST_j \in \mathcal{T}} Cnt_{i,j} \times wcet'_{i,j}$$

where $Cnt_{i,j}$ and $wcet'_{i,j}$ are the execution count and WCET of tick transition $ST_i \rightarrow ST_j$ in TTA $\mathcal{T}$.

The linear constraints on $Cnt_{i,j}$ are developed from the TTA's control flow. Since we are calculating the WCRT between an input signal and its output, only one of the initial transitions is allowed to take place. This is captured by the constraints

$$\sum_{ST_i \rightarrow ST_j \in \mathcal{T} \wedge ST_i \in Q_0} Cnt_{i,j} = 1$$

where $Q_0$ is the set of initial states of TTA $\mathcal{T}$. Furthermore, for each state $ST_i$ in the TTA, the aggregate execution counts of all incoming tick transitions should be equal to that of the outgoing

transitions. Thus,

$$\sum_{ST_j \rightarrow ST_i \in \mathcal{T}} Cnt_{j,i} - \sum_{ST_i \rightarrow ST_k \in \mathcal{T}} Cnt_{i,k} = 0$$

Over and above the constraints given in the preceding, we need to bound the number of iterations of every cycle in the TTA. This is a difficult task. The Esterel program may contain loops in one or more processes in the computation between the input signal *IN* and output *OUT*. Even if we know the loop bounds of these Esterel level loops, we cannot directly use them to bound the cycles of the TTA. While constructing the TTA from the Esterel program, an Esterel level loop is often partially unrolled, or fused with other loops.

We bound the number of executions of each TTA cycle as follows. Recall that each state in the TTA is a valuation of control state variables $s_1, \ldots, s_n$ – each variable $s_i$ corresponds to a thread or process in the Esterel program. Now, for each loop $L$ in the Esterel program, we first find the control state variable $s_k$ that captures the control flow of the process $p$ containing $L$. Since the loop defines repeated execution of certain local control states of the process $p$, and the variable $s_k$ simply encodes the progress in control flow of $p$ – we can always find a value $v$ of $s_k$ that appears exactly once in each iteration of the loop $L$. Such a value $v$ corresponds to a control state of $p$ lying inside the loop $L$. We now generate the following ILP constraints to incorporate the loop bound $B_L$ for each loop $L$ in the Esterel program

$$\sum_{ST_i \rightarrow ST_j \in \mathcal{T} \wedge (s_k == v) \in ST_i} Cnt_{i,j} <= B_L$$

where $s_k$ is the control state variable for the process containing loop $L$, $v$ is a value of $s_k$ that holds once in each iteration of $L$.

# 7. CASE STUDY

In this section, we first discuss some implementation details of our response time estimation framework (as shown in Figure 2).

In our experiments, an Esterel program is compiled into C by the Columbia Esterel Compiler (CEC). Instrumentation code is added into CEC to build a mapping between Esterel statements, intermediate representation's (e.g., abstract syntax tree, SCFG) node ID, and the generated C code. Chronos [9] – a static analysis based popular WCET estimation tool – is used to generate the WCET objective function, micro-architectural models and control flow constraints of the compiled C tick function ($ILP_{tf}$ in Figure 2). In this paper, we use the default architecture configuration of Chronos, which assumes a direct mapped L1 instruction cache, dynamic 2-level branch predictor, 5-staged pipeline, and an instruction dispatch queue size of 4. We consider an L1 instruction cache with 256 cache blocks, where each block's size is 8 bytes. In order to compute a tight WCET of each single tick in the TTA, constraints due to program level contexts are added into $ILP_{tf}$ to restrict the path can be taken in the tick function, which forms the ILP problem $ILP_{tick}$). $ILP_{tick}$ is solved by ILOG CPLEX to produce WCET value of each individual tick.

To illustrate our response time estimation technique, we used the TURBOchannel Interface (TcInt) benchmark from the Estbench Esterel Benchmark Suite [4]. It contains 684 lines of Esterel code and 2192 lines of compiled C code. TURBOchannel is an I/O interconnect that allows several I/O options to connect to one system [12]. In our case study, we consider the "ReadROM" operation which takes 9 ticks between the input event and the final output.

The WCET of the generated tick function is $10,949$ cycles without considering any tick path information from TTA. Since the operation takes a maximum of 9 ticks to finish, a pessimistic estimation of the response time is $9 \times 10,949 = 98,541$ clock cycles.

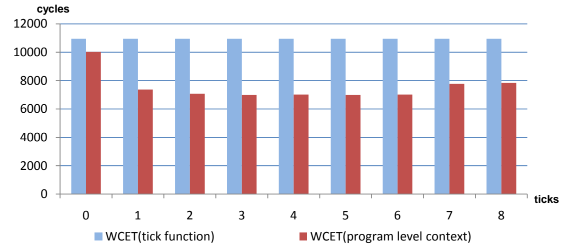


**Figure 3: Tick WCET results from different calculation approaches**

One the other hand, by utilizing program path contexts, the calculated response time is $68,103$ clock cycles, which gives a $30.9\%$ reduction over the previous result.

Finally, Figure 3 compares the WCET values calculated for the individual ticks which appear in the longest path of the TTA – ticks responsible for the worst-case response time of a ReadROM operation. For each tick, (i) the left bar shows the WCET value without any context information, (ii) the right bar shows the WCET value considering program level contexts. In our experiment, the ILP solving time for WCET of each individual tick takes 2-3 seconds, while ILP solving time to compute WCRT of multiple ticks takes less than 0.1 second.

# 8. CONCLUDING REMARKS

In this paper, we have proposed a context-sensitive timing analysis for Esterel programs, which is useful for obtaining tight estimates on the response times of input events being processed by the program. Towards this goal, we captured control flow contexts at the beginning of each Esterel clock tick.

In future, we will also consider micro-architectural context (e.g., cache states) to obtain even tighter WCET estimates. Note that we have assumed a single-core processor architecture in this paper. It would be interesting to investigate possible extensions of this work to multi-cores.

# 9. REFERENCES


[1] AbsInT GmbH, http://www.absint.com/.

[2] M. Boldt, C. Traulsen, and R. von Hanxleden. Worst Case Reaction Time Analysis of Concurrent Reactive Programs. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 203(4):65–79, 2008.

[3] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 9(79):1270–1282, 1991.

[4] S.A. Edwards. The Estbench Esterel Benchmark Suite. *http://www1.cs.columbia.edu/ sedwards/software.html*, 2003.

[5] S.A. Edwards and J. Zeng. Code Generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems*, 2007.

[6] A. Benveniste *et al.* The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[7] R. Heckmann *et al.* Combining a high-level design tool for safety-critical systems with a tool for WCET analysis on executables. In *4th European Congress on Embedded and Real Time Software (ERTS)*, 2008.

[8] V. Bertin *et al.* TAXYS = Esterel + Kronos. A tool for verifying real-time properties of embedded systems. 2001.

[9] X. Li *et al.* Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3), 2007. http://www.comp.nus.edu.sg/~rpembed/chronos.

[10] L. Ju, B.K. Huynh, A. Roychoudhury, and S. Chakraborty. Performance debugging of Esterel specifications. In *International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS)*, 2008.

[11] G. Logothetis, K. Schneider, and C. Metzler. Generating formal models for real-time verification by exact low-level runtime analysis of synchronous programs. In *RTSS*, 2003.

[12] M.J.K. Nielsen. TURBOchannel. In *36th IEEE Computer Society International Conference, COMPCON*, 1991.

[13] D. Potop-Butucaru, S.A. Edwards, and G. Berry. *Compiling ESTEREL*. Springer, 2007.

[14] T. Ringler. Static worst-case execution time analysis of synchronous programs. In *5th Ada-Europe International Conference*, LNCS 1845, 2000.

[15] R. K. Shyamasundar and J. V. Aghav. Realizing real-time systems from synchronous language specifications. In *RTSS Work-in-Progress Session*, 2000.