Proceedings of the
15th International Workshop on
Automated Verification of Critical Systems (AVoCS 2015)

Model-based WCET Analysis with Invariants

Bojan Nokovic, Emil Sekerinski

15 pages
Model-based WCET Analysis with Invariants

Bojan Nokovic, Emil Sekerinski

{nokovib, emil}@mcmaster.ca, McMaster University, Hamilton, Ontario, Canada

Abstract: The integration of worst case execution time (WCET) analysis in model-based designs allows timing problems to be discovered in the early phases of development, when they are less expensive to correct than in later phases. In this paper, we show how model-based WCET analysis can improve timing calculations compared to program-based WCET analysis. The models are described by hierarchical state machines with concurrency, probabilistic transition, stochastic transitions, costs/rewards attached to states and transitions, and invariants attached to states. In these models, user-specified invariants serve to check the correctness of designs by restricting allowed state configurations. Our contribution is to use invariants additionally to determine transition combinations (paths) that can be eliminated from the WCET analysis, with the help of a decision procedure, thus making the analysis more precise. The assembly code of transitions for a specific target is generated and execution time for that code calculated. From the model, a probabilistic timed automaton (PTA) or Markov decision process (MDP) can be created. On that model, execution times of transitions are calculated as costs.

Keywords: Hierarchical state-machines, Compiler, Worst-case analysis

1 Introduction

The full integration of embedded systems into larger products leads to an increased reliance on their correct service: in embedded systems, not only safety and liveness, but also strict timing constraints must be satisfied [LM11]. Properties of a program like loop bounds, exponential path space, path feasibility, and properties of the hardware like memory access time, determine program execution time. An established approach is to insert constraints as deadlines in basic blocks of the source code and, after compiling, the assembly code is used in a low-level analysis to determine execution time [LS14]. If the calculated times are less than or equal to the specified deadlines, the timing constraints are satisfied.

Following the trend that part of modern software engineering is the integration of worst case execution time (WCET) analysis [KP05], we consider model-based WCET calculation for early, automatic analysis. The models are expressed as pCharts, a formalism for reactive systems based on hierarchical state machines with invariants, probabilistic transitions, timed transitions, stochastic timing, and costs/rewards [NS13, NS14]. The pCharts formalism underlies pState, a tool for the specification, design, qualitative analysis, quantitative analysis, and implementation of embedded systems [Nok15]. With pState users can specify what a system does; from the specified structure the tool generates executable code which determines how the system works, and finally by generating input code for a model checker it is possible to reason why the system does what it does (or fails at what it is specified to do).
We show that state invariants, intended to be written by developers to capture design decisions for correctness (qualitative) analysis, can also be used to improve WCET analysis. The bound for the execution time can be specified directly on a transition and is automatically checked for feasibility. If the calculated WCET is greater than the bound, the design has to be alternated or the design target updated by selecting a different processor, increasing the clock of the processor, etc.\footnote{Examples of automatic calculation of WCET from pCharts models can be found at http://pstate.mcmaster.ca/} Thus this allows a design process in which low-level timing considerations can impact the design. The calculation of WCET takes the run-time overhead of scheduling and cancelling timed transitions into account. The advantage of this holistic modelling approach is the simplicity gained by having only a single model and relying on automated code generation, without the need for model transformations.

Statecharts \cite{Har87}, hierarchical state machines with concurrency and broadcasting are a graphical specification formalism for reactive systems, but they are also executable and compilable \cite{Har07}. Similarly to iState \cite{SZ01}, pState implements an event-centric semantics in which external events trigger immediate execution, unlike the state-centric semantics of UML and statecharts, in which events are data (in queues). These two interpretations are called requirements-oriented and implementation-oriented semantics in \cite{EJW02}. For example, the TCM toolkit for conceptual modelling also follows the requirements-oriented semantics \cite{DW03}. The event-centric semantics simplifies the correctness analysis, as every transition is an atomic step, even in presence of broadcasting. The semantics is suitable for embedded systems where events are processed quickly enough and queuing of events is not desirable or not possible, e.g. with micro-controllers with limited memory.

Timing analysis is divided into (1) analyzing control-flow properties and (2) calculating execution time of instructions or basic blocks of instructions \cite{LS14}. From a pCharts model, pState generates code and automatically calculates the execution time of instructions. To this end, bodies of transitions are normalized to nested if-then-else statements with multiple assignments, and a satisfiability modulo theories (SMT) checker, Yices \cite{Dut14}, is used to analyze control-flow properties, with the use of invariants. The number of execution cycles is then calculated for each basic block. The calculation of the upper bound of the execution time of programs is not possible in general (the halting problem). Currently pState imposes syntactic restrictions (no general while-loops and recursion) such that transitions always terminate and automatic calculation of the upper bound is possible.

State invariants are conditions that are attached to states in a state hierarchy and specify what has to hold in that state. With hierarchical and concurrent states, the accumulated invariant of a state is obtained by “inheriting” the invariants of parent states \cite{Sek08}. Every incoming transition to a state must ensure that its accumulated invariant holds and every outgoing transition can assume that the accumulated invariant holds. This gives a method for checking a chart against an invariant annotation. State invariants can express safety of an embedded system or consistency of a software system. Invariants used in this way are a kind of flow facts annotations, additional information about control flow \cite{KP05}: invariants are explicitly specified, rather than automatically derived, but are checked separately with a decision procedure, hence can assumed to be correct. Invariants were used before in finding unfeasible paths, e.g. see \cite{HGB08,EES01,GJK09} for overviews. Our contribution here is that we do not rely on the user suggesting or
Figure 1: Every 10ms, state Time broadcasts event tick and transitions on tick in PortA and PortB are executed simultaneously.

A tool automatically trying to derive invariants for the purpose of finding unfeasible paths, but we use model-based invariants which users would have stated for checking the correctness of models. If a stated invariant is not preserved by all affected transitions, it is ignored for unfeasible path analysis. This extends to invariants in models with concurrency and broadcasting. The specific benefit of this approach is that invariants can be used in early WCET calculation, without need for data flow analysis. As a simple example, in Fig. 1 the invariant

\[ \text{in PortA.Off} \lor \text{in PortB.Off} \]

states that PortA is in Off, or PortB is in Off, or both are in Off. They can not be in Through simultaneously, therefore the code will never execute tick transitions in both PortA and PortB at the same run. This is not the strongest invariant; a stronger invariant would be

\[ \text{in PortA.Off} \oplus \text{in PortB.Off} \]

where \( \oplus \) is exclusive OR. For the execution time calculation of the tick event, there is no difference if we use either one, as both invariants exclude the possibility that PortA and PortB are in Through at the same time. The event tick is generated by a timer periodically, every 10ms. The time for executing both transitions on tick should not contribute to the WCET of transition t. Informally, in this case:

\[ WCET(t) = \max(\text{WCET}((10ms/tick) \parallel (\text{tick}/\text{readA})), \text{WCET}((10ms/tick) \parallel (\text{tick}/\text{readB}))) \]

Without the invariant, WCET of the transition t would be calculated as:

\[ WCET(t) = \text{WCET}((10ms/tick) \parallel (\text{tick}/\text{readA}) \parallel (\text{tick}/\text{readB})) \]
Transition conditions under invariants are checked by the SMT solver Yices as the backend tool. When broadcasting to concurrent states, all combinations of executable paths are created. If some conditions are never satisfied, the associated paths are excluded.

We start with preliminaries and an overview of the translation scheme in Sec. 3.1 and 3.2. In Sec. 3.3 we show how conditions can be automatically verified and unreachable paths removed from the executable code. Besides simplifying the generated code, this makes the WCET calculation more accurate. We illustrate in Sec. 3.4 how user-specified state invariants improve WCET calculation. Accurate WCET analysis can be done only at assembly/object code level; therefore our nested hierarchical state models have to be translated into assembly code. A brief description of this process is given in Sec. 4.

2 Related Work

We consider static, or verification-based WCET calculation. The upper bound of the task execution is estimated on the code itself taking the hardware architecture into account. Static analysis guarantees that execution time will not exceed the upper bound, but sometimes this estimation may be too pessimistic, which can be confirmed by measurement-based methods. Other techniques for WCET calculations are simulations and path analysis for the calculation of execution scenarios [KP05]. In general, methods of static program analysis include value analysis, control flow analysis (CFA), processor behaviour analysis, and symbolic simulation. Some well-known WCET tools for static program analysis are aiT, Bound-T, OTAWA, and SWEET [WEE+08].

A method for analyzing real-time behaviour of reactive synchronous system with special focus on statecharts is described in [EA99]. In addition to WCET and schedulability analysis of statecharts models, worst case response time (WCRT) of synchronous models is introduced. WCRT includes possible interference by other programs. The method for calculation of WCET/WCRT is implemented in STATEMATE [HN96].

A difference between UML State Machine Diagrams [Fow03] and pCharts is related to the specification of internal state activities. UML allows two types of internal state activities, do-activities, and regular activities specified by entry and exit keywords. Regular activities occur “instantaneously” while do-activities can take finite time and can be interrupted. pCharts do not allow the explicit specification of internal state activities, but the entry part of the activity can be specified in the body of incoming transitions, and exit part can be specified in the body of outgoing transitions. The execution of a transition in pCharts is instantaneous, like in UML, meaning it cannot be interrupted by another event. In pCharts, if a transition is enabled through broadcasting an event, also transitively, all transitions take place simultaneously, unlike in UML.

The UML Profile for Modelling QoS provides facilities for defining a wide variety of QoS requirements and properties [KJ10], divided into categories: performance, dependability, security, integrity, coherence, throughput, latency, efficiency, demand, reliability, and availability. In pCharts, a transition deadline can be specified directly on the model: the specified deadline is similar to the QoS latency property, since both refer to a time interval during which a response to an event must be completed. Other properties like performance, dependability, throughput are expressible in pCharts models as well using timed and stochastic transitions and costs/rewards attached to states and transitions.
The behavioural constructors used in the OMG system modelling language SysML consist of activity diagrams, sequence diagrams, state machine diagrams, and use case diagrams [OMG15a]. State machine diagrams in SysML are based on the standard UML state machine concept, so do not support state invariants.

The UML profile for real-time systems (UML-RT) extends the basic UML concepts to facilitate the design of complex real-time systems [KJ10]. UML-RT is an industrial standard and can be used to design event-driven real-time systems. The standard is primarily focused on architecture specification of real-time systems, but the behaviour can be modelled by state machine diagrams. UML-RT state machines do not support nested concurrent states and automatic WCET calculation whereas pState does.

A process of model-based design which guarantees hard real-time requirements is described in [CBC08]; that approach is based on Time Petri Nets rather than state machines. Model-based tools can integrate other specialized tools. For instance, for determination of resource usage and timing analysis, modelling tool Esterel SCADE Suite [Tec15] uses StackAnalyzer and aiT.

Timed-automata and corresponding model checkers like UPPAAL have been used for WCET calculation [BC11]. METAMOC reduces computing of WCET to finding the longest path of the CFG represented as timed-automaton [DOT+10]. Model checking has also been used for low level analysis dealing with caches and pipelines [Met04]. User-specified invariants have not been considered in any of these approaches.

Using model checking, but not dealing with caches and pipelines, pState first predicts the timing of program instruction for the specified hardware, and then the time of the specified path is calculated by a model checker. For basic blocks, the number of processor ticks is introduced as the cost of a transition. The execution time is calculated as the cost on a specified path.

The MARTE UML profile [OMG15b] for modelling and analysis of real-time and embedded systems is intended to replace the existing UML Profile for Schedulability, Performance and Time (SPT) [KJ10]. Behaviour scenarios are annotated with expressions of describing some non-functional properties (NFP), like $\text{NFP}_{\text{Duration}} = \text{wcet}$. The annotated model indicates to the analysis tools what property has to be computed. While WCET calculation is part of the MARTE extension, we are not aware that any of the tools that implements MARTE profile also take into account invariants when calculating the WCET.

The work of [RMPC13] recognizes that when generating programming language code from models and analyzing the WCET of the code, high-level information present in models is not available to the WCET analyzer. Using a synchronous data flow language for modeling, the approach is to supply information about infeasible paths to the WCET analyzer by computing invariants that exclude those paths; the invariants are checked with a dedicated model checker. In pState, invariants are integral part of the design with pCharts, rather than being computed.

The application area for which pState was originally developed are embedded systems with 8 and 16 bit micro-controllers, e.g. as used in active RFID tags [Pau06]. Those applications have a small number of states, so the problem of scalability of model checkers is not a primary concern. Data processing elements like prefetching, delayed branching, and branch prediction, which can complicate WCET on advanced microprocessors, are not present in those simple micro-controllers. We assume that the execution time of the instruction is independent of the instruction order. Code generation and WCET calculation is integrated in pState and sufficiently fast to allow interactive WCET analysis.
3 Model-based WCET

3.1 Preliminaries

Every event is translated to an intermediate representation consisting of `skip`, assignments, `if-then-else` statements, and parallel (independent) compositions (which arise from broadcasting). Following rules are used to eliminate parallel composition. Let $b$ be boolean expressions, $Q$, $R$, $S$ statements, $x$, $y$ variables, $e$, $f$ expressions. The transformation rules are:

\[
\begin{align*}
  x := e & \parallel y := f = x,y := e,f & \quad (1) \\
  \text{if } b \text{ then } Q & = \text{if } b \text{ then } Q \text{ else skip} & \quad (2) \\
  (\text{if } b \text{ then } Q \text{ else } R) \parallel S & = \text{if } b \text{ then } (Q \parallel S) \text{ else } (R \parallel S) & \quad (3) \\
  Q \parallel R \parallel S & = Q \parallel R \parallel S & \quad (4) \\
  Q \parallel \text{skip} & = Q & \quad (5)
\end{align*}
\]

These rules allow to transform a statement to nested `if-then-else` statements with the innermost statements being multiple assignments. That form is used for WCET analysis, as below.

3.2 Transition with Specified Deadline

Figure 2 shows a transition from state $S$ to state $T$ on event $E$. Guard $x = 0$ represents the condition which must be satisfied for the transition to be taken, body $x := x + 1$ is the action which is executed when the transition is taken, and $20\mu s$ is the specified deadline for that transition. Guard, action, and deadline are optional in charts. The operation $op(E)$ on event $E$ returns a set of prioritized guarded commands, with nondeterminism among transitions on the same event, but outer transitions taking priority over inner transitions, according to the algorithm in [NS14]. The nondeterminism can be arbitrarily resolved, such that $op(E)$ becomes a nested `if-then-else` statement. Here $op(E)$ is given by:

\[
op(E) = \text{if } in(S) \land (x = 0) \text{ then } \text{goto}(T) \parallel x := x + 1 \text{ else skip}
\]

To check if the WCET of the transition from $S$ to $T$ takes at most $n$ cycles, we need to calculate the execution time on $E$ by translating $op(E)$ into assembly code. If the target is the RISC PIC16F6xx micro-controller, the code is in PIC assembly language, which we call `picasm`. The instruction set has about 35 instructions divided into three groups, byte-oriented, bit-oriented and control operations. Most of the instructions take one processor cycle, except jump and subroutine call which take two cycles. The operation $op(E)$ can be executed if the chart is in state $S$, written as $in(S)$ and guard $x = 0$ holds. The effect of the transition is execution of the then branch $x := x + 1$. The chart is going to state $T$, written as $\text{goto}(T)$. With a simple translation scheme employed, the WCET of $op(E)$ can be calculated compositionally:
Listing 1: Generated assembly code for \textit{op}(E)

\begin{verbatim}
picasm(op(E)) =
movf r, W ; W := r
xorlw S ; W := W XOR S
btfss STATUS, 0x2 ; If Z = 1 skip
goto CONTINUE

GUARD_0
movf x, W ; W := x
xorlw 0 ; W := W XOR 0
btfss STATUS, 0x2 ; If Z = 1 skip
goto CONTINUE

ACTION_0
movlw 1 ; W := 1
addwf x, 1 ; x := W + x
movlw T ; W := T
movwf r ; r := W

CONTINUE
\end{verbatim}

\[ WCET(picasm(op(E))) = WCET(picasm(in(S))) + WCET(picasm(x = 0)) + WCET(picasm(goto(T) \parallel x := x + 1)) \]

The composition $goto(T) \parallel x := x + 1$ is well-defined only if the variables assigned by $goto(T)$ and by $x := x + 1$ are disjoint. Sequentialization of multiple assignments may need extra variables for temporary storage. The time to copy values into these variables has to be taken into account. The sequentialization of multiple assignments is done with \textit{minimum} number of auxiliary variables. Statement $goto(T)$ is here defined as $r := T$, where $r$ is the \textit{root} state in which $S$ and $T$ are nested. Therefore this parallel composition is well defined and there is no need for an extra variable for sequentialization, so

\[ (goto(T) \parallel x := x + 1) = (r := T; x := x + 1) \]

and \textit{WCET} of this parallel composition is

\[ WCET(picasm(goto(T) \parallel x := x + 1)) = WCET(picasm(r := T)) + WCET(picasm(x := x + 1)) \]

The generated assembly code is shown in Listing 1. The number of cycles for \textit{in}(S) is 3, etc., and the \textit{WCET} of \textit{op}(E) calculated by \textit{pState} as

\[ WCET(picasm(op(E))) = 3 + 3 + 2 + 2 = 10 \text{ cycles} \]

One processor \textit{cycle} for micro-controller RISC PIC16F6xx running at 4MHz is 1\,\mu s. In Figure 2, we specify $\Delta = 20\mu s$, which is greater than the \textit{WCET} of 10\,\mu s, so we can say that the transition on event $E$ satisfies the specified requirement for this particular target. To simplify notation, we leave out the translation to assembly if understood from the context: \textit{WCET}(S) stands for \textit{WCET(target_asm}(S)) for statement $S$. 

The composition $goto(T) \parallel x := x + 1$ is well-defined only if the variables assigned by $goto(T)$ and by $x := x + 1$ are disjoint. Sequentialization of multiple assignments may need extra variables for temporary storage. The time to copy values into these variables has to be taken into account. The sequentialization of multiple assignments is done with \textit{minimum} number of auxiliary variables. Statement $goto(T)$ is here defined as $r := T$, where $r$ is the \textit{root} state in which $S$ and $T$ are nested. Therefore this parallel composition is well defined and there is no need for an extra variable for sequentialization, so

\[ (goto(T) \parallel x := x + 1) = (r := T; x := x + 1) \]

and \textit{WCET} of this parallel composition is

\[ WCET(picasm(goto(T) \parallel x := x + 1)) = WCET(picasm(r := T)) + WCET(picasm(x := x + 1)) \]

The generated assembly code is shown in Listing 1. The number of cycles for \textit{in}(S) is 3, etc., and the \textit{WCET} of \textit{op}(E) calculated by \textit{pState} as

\[ WCET(picasm(op(E))) = 3 + 3 + 2 + 2 = 10 \text{ cycles} \]

One processor \textit{cycle} for micro-controller RISC PIC16F6xx running at 4MHz is 1\,\mu s. In Figure 2, we specify $\Delta = 20\mu s$, which is greater than the \textit{WCET} of 10\,\mu s, so we can say that the transition on event $E$ satisfies the specified requirement for this particular target. To simplify notation, we leave out the translation to assembly if understood from the context: \textit{WCET}(S) stands for \textit{WCET(target_asm}(S)) for statement $S$. 

\[ WCET(picasm(op(E))) = 3 + 3 + 2 + 2 = 10 \text{ cycles} \]
3.3 WCET of Parallel Composition

Figure 3 shows a transition from state $T_1$ to $T_2$ on event $E$. Suppose

$$stat_i \equiv \text{if } c_i \text{ then } S_i, \quad c_1 \equiv x > 0, \quad c_2 \equiv x < 0$$

where $S_i$ are statements. Then:

$$\text{op}(E) = \text{if in}(T_1) \text{ then goto}(T_2) \parallel stat_1 \parallel stat_2 \text{ else skip}$$

The parallel composition $stat_1 \parallel stat_2$ amounts to:

$$\left(\text{if } c_1 \text{ then } S_1\right) \parallel \left(\text{if } c_2 \text{ then } S_2\right)$$

= by (2)

$$\left(\text{if } c_1 \text{ then } S_1 \text{ else skip}\right) \parallel \left(\text{if } c_2 \text{ then } S_2 \text{ else skip}\right)$$

= by (3)

$$\left(\text{if } c_1 \text{ then } (S_1 \parallel S_2 \text{ else skip}) \text{ else (skip \parallel if } c_2 \text{ then } S_2 \text{ else skip)} \right)$$

= by (5) and symmetry of $\parallel$

$$\left(\text{if } c_1 \text{ then } (S_1 \parallel S_2 \text{ else skip}) \text{ else (if } c_2 \text{ then } S_2 \text{ else skip)} \right)$$

= definition of $c_i$

$$\text{if } x > 0 \text{ then (if } x < 0 \text{ then } S_1 \parallel S_2 \text{ else } S_1 \text{ else (if } x < 0 \text{ then } S_2 \text{ else skip)} \right)$$

Now, if the SMT solver verifies that $x > 0 \land x < 0$ is unsatisfiable and $S_i$ do not assign to common variables, above is equivalent to

$$\text{if } x > 0 \text{ then } S_1 \text{ else (if } x < 0 \text{ then } S_2 \text{ else skip)}$$

and the WCET of $\text{op}(E)$ is calculated by $p\text{State}$ as:

$$\text{WCET}(\text{op}(E)) = \text{WCET}(\text{in}(T_1)) + \text{WCET}(\text{goto}(T_2)) + \text{WCET}(x > 0) + \max(\text{WCET}(S_1), \text{WCET}(x < 0) + \text{WCET}(S_2))$$

That is, infeasible paths are excluded. The calculation is conservative: if the SMT solver fails to establish that the conjunction of guards is unsatisfiable, the WCET of the code with all paths is taken.
3.4 WCET Taking State Invariants into Account

State invariants are used to improve WCET calculation by providing additional information, which makes the analysis feasible and tight. Figure 1 presents the process of reading redundant digital ports. If we assume that our target is PIC16F636, we define the action readA as reading port A into predefined variable ra by \( ra := PORTA \), where \( PORTA \) is port A of the micro-controller. Similarly, we define reading from port B. If the signal is lost on port A, the event toggle is generated and the system starts reading data from port B. From the generated code, the WCET of transition in state Timer is automatically calculated as 10ms + 160 cycles. However, when PortB is in Through, PortA must be in Off and vice versa. So, it is not possible to read both ports A and B at the same tick, but this is so far not taken into account when the WCET is calculated. By adding the invariant \( in PortA.Off \lor in PortB.Off \) we specify that PortA and PortB can not be in the Through states at the same time. pState verifies this automatically and excludes one of the transition on tick when calculating the WCET. The calculated WCET is now 10ms + 148 cycles. The complete generated assembly code as used for WCET calculation is at http://pstate.mcmaster.ca/.

3.5 WCET Taking into Account Timed Transitions

We consider two timed transitions \( k \) and \( l \) as in Fig. 4. Transition \( k \) takes place exactly \( t_1 \) time units after the state \( S \) is entered, if event \( E \) does not occur in the meantime. Similarly, transition \( l \) happens \( t_2 \) timed units after state \( T \) is entered. The operation on event \( E \) for the transition in Fig. 4 includes cancelling timed transition \( k \) and scheduling timed transition \( l \) is

\[
\text{op}(E) = \begin{cases} 
\text{if } in(S) \land g \text{ then } (\text{goto}(T) \parallel a); \text{ cancel } k; \text{ schedule } l \text{ else skip} 
\end{cases}
\]

For the WCET of \( E \) we need to add the time of scheduling transition \( l \) and the time to cancel transition \( k \). Again, the transition is considered to be correct if the WCET is less or equal to specified deadline \( d \), i.e. \( WCET(\text{op}(E)) \leq d \). A scheduler for timed transitions is created separately and is called from the generated code for scheduling and cancelling transitions. Scheduling of timed transitions involves adding new events into a data structure of scheduled events, sorted by time and priority. Procedure cancel contains a loop to search for the scheduled timed transition which has to be removed. When the first due-to event is removed, other events are shifted down. In the current implementation of the scheduler, the time to schedule an event if the data structure is empty is 52 processor cycles. The time to go trough the loop and sort events is 141 cycles per event. The WCET of scheduling is less then or equal to the number of scheduled transitions multiplied by 141 plus 52 cycles. One run through the loop of a canceled timed transition takes 35 cycles, while the time to shift the scheduled event takes 75 cycles. In a similar way, we have that
the WCET of cancelling a transition is less than or equal to the number of scheduled transitions multiplied by 35 plus 75 cycles.

\[ WCET(\text{cancelling of timed transition}) = 35 \times \# \text{ of scheduled transition} + 75 \]

For the transition on event \( E \) from Fig. 4, \( pState \) calculates

\[ WCET(\text{op}(E)) \leq 10 + 52 + 35 \times 1 + 75 = 172 \text{ cycles} \]

Note: The number processor cycles are related to the current implementation. Any modification of the algorithm may have an impact on those numbers.

### 3.6 WCET of Timed Transition

Consider timed transition \( t \) in Fig. 5. The operation \( \text{top} \) on timed transition \( t \) is given by

\[ \text{top}(t) = \text{if } \text{in}(S) \text{ then goto}(T) \text{ else skip} \]

The transition takes place 100 ms after state \( S \) is entered. To calculate the WCET we need to add the time required by the scheduler to start this transition.

\[ WCET(\text{top}(t)) = WCET(\text{goto}(T)) + WCET(\text{scheduler cycle}) \]

The scheduler uses TIMER0 of the target processor to generate an interrupt every 1 ms. In the interrupt service routine, the due-time for each scheduled event is decreased. This is done in a loop and the execution time depends of the number of scheduled events. Also, in each scheduler cycle, possible events need to be polled.

\[ WCET(\text{scheduler cycle}) = WCET(\text{decrease due time}) + WCET(\text{event polling}) \]

Based on the current implementation, the time to decrease the event due-time is 98 cycles if there is only one task scheduled. Polling events has two parts, polling of timed transitions and polling of input events on the micro-controller’s ports:

\[ WCET(\text{event polling}) = WCET(\text{polling timed transitions}) + WCET(\text{polling input events}) \]

The WCET for \( \text{polling timed transitions} \) is 22 cycles, while the WCET of \( \text{polling input events} \) is only 5 cycles plus the time to execute the input event.

\[ WCET(\text{event execution}) \leq 22 + \sum \{ 5 + \text{input event } I \text{ execution} | \text{input event } I \} \]

For timed transition \( t \), taking into account calculations from Sec. 3.2, and assuming that (1) there is only one event to execute, and (2) there are no input events, \( pState \) calculates the WCET as:
\[ WCET(picasm(op(t))) = 100\text{ms} + 3\text{cycles [in(S)]} + 2\text{cycles [goto(T)]} + 98\text{cycles [tick]} + 22\text{cycles [event execution]} \]

\[ = 100\text{ms} + 125\text{cycles} \]

If the target micro-controller runs on 4MHz, the execution time of one cycle is \(1/(4\text{MHz}/4)\) which is \(1\mu s\). In this case, \(WCET(op(t)) = 100.125\text{ms}\). If the specified deadline is \(\Delta \geq 125\mu s\), the transition execution time is satisfiable. But, if we need \(\Delta \geq 100\mu s\), we will find out during the specification phase that the requirement can not be implemented. In that case, a possible solution is to select a higher frequency of micro-controller clock. Instead of 4MHz, we can use 8MHz crystal clock, which will fix the problem. By this approach in the early phase of design during specification we can identify some hardware limitations and make appropriate design decisions.

The operation \(top(t)\) of the timed transition in Fig. 5 includes code that checks the source state, \(S\), before it takes the transition to another state. As the transition is only scheduled when the chart is in \(S\), that check is not needed, but is still included to protect against faults like incorrectly scheduled events.

Long-running real-time systems are known to be prone to cumulative clock drifts. This can occur if the scheduling of new timed transitions is delayed by the time it takes to schedule the new transition. Cumulative drift is avoided here by having TIMER0 generate periodically an interrupt every 1\(\text{ms}\) and requiring that all transitions are completed within 1\(\text{ms}\). That is, every transition, regular and timed, has an implicit deadline of that corresponds to the timer resolution.

### 4 Assembly Code Generation

Assembly code is created by translating the intermediate code representation. The translation of if-then-else statements is straightforward. The translation of parallel statements needs extra processing since these have to be converted into sequential compositions as shown in Secs. 3.2 and 3.3. The multiple assignment \(v_1, v_2 := e_1, e_2\) is translated to \(v_1 := e_1; v_2 := e_2;\) if \(v_1\) does not occur in \(e_2\). In general, for the sequentialization of the multiple assignment

\[ v_1, \ldots, v_n := e_1, \ldots, e_n \]

we may need to create one or more extra variables. The problem of sequentializing multiple assignments can be expressed as follows: for given \(v_i (i \in 1..n)\) and expressions \(e_i\) which are only dependent on these variables, we define the dependency relation to be a directed graph \(G = (N,E)\), where \(N = \{n_i | i \in 1..n\}\), and

\[ E = \{(n_i, n_j) | i \in 1..n \land j \in 1..n \land e_i \text{ is dependent on } e_j\} \]

When \(G\) is acyclic, the sequentialization is trivial. When there are one or more cycles in \(G\), extra variables are necessary to eliminate the cycles. Introducing an extra variable for \(v_i\) will remove \(n_i\) and all connected edges from \(G\). A cycle in \(G\) is broken when any \(n_i\) in the cycle is removed. One node can be in more than one cycles; if we call the number of each \(n_i\) the cycle degree \(c_i\), then removing \(n_i\) would break these \(c_i\) cycles in \(G\). Therefore, sequentialization of multiple assignment can be reduced to the process of removing cycles form the feedback vertex set graph.
That is proven to be an NP-complete problem [GJ90]. Because of this, our implementation of sequentialization provides an approximate solution. Based on experimentation, for most practical cases where \( n \leq 5 \), the approximation provides an optimal solution.

Chart states are generated as enumeration names of states, and variables as integers. In charts, integer variables are declared as subrange types with lower and upper bound, but the target micro-controller allows only subranges that fit into one byte. In the generation of assembly code, the technique of delayed code generation [Wir96] is used, which produces optimal addressing modes and register usage for this simple architecture in a single pass. This makes code generation sufficiently fast that WCET analysis can be done interactively.

Scheduler, initialization and I/O actions are not generated from the specification, they are write-once code. In this way we have full control over the structure of the application, similar to the approach described in [IAR99].

5 Summary, Conclusion, and Future Work

We implemented a framework for model-based WCET analysis of real-time systems. From a hierarchical representation, executable code in a low level language is generated. On the generated code, WCET of transitions can be calculated by counting the number of assembly instructions execution cycles. The precise WCET determination on complex architecture is a challenging problem, but determining the WCET on 8 and 16 bit micro-controllers is easier since features like multi-stage pipelines and caches are not present. Invariants for improving the WCET are independent of the architecture of the target processor. The \( pState \) architecture allows in principle external tools for WCET to be plugged in. Existing WCET analyzers like AbsInt [Inf15] could calculate the WCET of executable code generated by \( pState \), but the invariants need to be stated in the AbsInt general annotation file; the annotation could in principle be generated by \( pState \) as well.

In order to exclude infeasible paths, conditions specified by state invariants and conditions created by sequentialization of parallel compositions are verified by the SMT solver Yices. In our models we do not allow general while loops and recursion, so the calculation of WCET is possible. Actions can contain external calls, e.g. to I/O libraries, that are out of control of \( pState \): for those timing constraints cannot be specified. Thus, if loops or recursion are necessary, these can be implemented by an external call. We plan to work on allowing loops to be expressed directly in \( pState \).

Worst-case-response-time (WCRT) that includes the impact of other transitions is especially important in concurrent systems. Invariants could be taken into account in a similar way as they are taken for calculation of WCET. Latency is the time passed from the moment when the event is registered to the moment when execution of the event starts. It depends on the implementation of the scheduler and I/O synchronization. The automatic calculation of transition WCRT and the estimation of the latency are the focus of future work.
Bibliography


