Abstraction Techniques for Compositional State-based Scheduling Analysis

Tayfun Gezgin, Stefan Henkler
Institute for Information Technology (OFFIS)
Oldenburg, Germany
{gezgin, stefan.henkler}@offis.de

Achim Rettberg, Ingo Stierand
Carl von Ossietzky University Oldenburg
Oldenburg, Germany
{achim.rettberg.stierand}@informatik.uni-oldenburg.de

Abstract—Nowadays, most embedded safety critical systems have to work in a timely manner in order to deliver desired services. In such timed systems not only ordering of events but timing properties are relevant for correctness and performance. In order to be safe and reliable, it is important to have rigorous analysis techniques of timing-dependent (state) behavior. Classical scheduling approaches consider only the system behavior stateless. Especially for safety critical systems this is not sufficient as the state space gives important information of the system which has to be considered by analysis approaches. Our approach for scheduling analysis combines analytical and model checking methods. We consider not only critical instances but the full state space for analysis, where all inter-leaveings and task dependencies are preserved. For this, the state space of the entire system architecture is constructed with the aid of input event streams for tasks, and the known behavior of the scheduler of each resource. Based on the state space response times can be determined, and safety properties can be verified by means of reachability checks. As this approach alone is not scalable we present abstraction techniques based on determining output event streams for each resource. For this we exploit well known analytical methods for scheduling analysis. These methods typically abstract from all inter-leaveings leading to very pessimistic results. In this work we present an abstraction technique that is relevant if multiple activations of one task can occur. This technique lies in the middle of both approaches mentioned above.

Keywords—Embedded Systems; Real-Time Systems; Scheduling Analysis; Model Checking; Abstraction Techniques

I. INTRODUCTION

Developing safety critical embedded systems has become an ever more complex task in recent years. An increasing number of system functions which are realized by software, inter-dependencies of software tasks and the integration of legacy systems lead to highly complex software intensive systems. For safety critical systems it is crucial that they adhere to their specifications, as the violation of a requirement could result in critical situations leading to very high costs or even threats to human life. Also the validation of requirements in early design steps is a critical issue, as late changes typically lead to high costs. One crucial aspect for safety critical systems is that they have to work in a timely manner. By choosing too slow computation units due to savings of cost, the system could react too slow in some situations leading to critical scenarios. So, it is important to have rigorous analysis techniques of timing-dependent behavior in order to develop safe and reliable systems.

Basically two approaches exist for scheduling analysis of distributed real-time systems. The classical approach is a holistic one, as it was worked out by e.g. Tindell and Clark [14]. Here, local analysis is performed evaluating fixed-point equations. The analysis is very fast and is able to handle large systems evaluating performance characteristics like time and memory consumption. Unfortunately, it delivers very pessimistic results when inter-ECU task dependencies exist. Furthermore, complex functional aspects cannot be analyzed with this approach like for instance safety properties, or more generally, reachability of certain states.

In [12] activation pattern for tasks are described by upper and lower arrival curves realizing a more compositional analysis method. Based on this work and classical real time analysis works, compositional scheduling analysis, called SymTaS, was realized by SymtaVision. The concept has been developed by Kai Richter et.al. [10], and was improved and extended in several works, e.g. [11]. The main idea behind SymTaS is to transform event streams whenever needed and to exploit classical scheduling algorithms for local analysis. This concept is very fast and is able to handle large systems, but typically yields pessimistic results.

The second approach is based on model checking, and has been illustrated for example in [7], [5]. Here, all entities like tasks, processors, and schedulers are modeled in terms of timed automata. The advantage of this approach is that one gets exact solutions with respect to the modeled scheduling problem. As the state space of the analyzed system is preserved, checking complex characteristics besides timing, like safety properties is possible. Unfortunately, the approach is not scalable, as the state space of the whole architecture is generated in one step and is tried to be checked directly.

Our approach for scheduling analysis combines both analytical and model checking methods. Analogous to [7] we consider the full state space for analysis, where all inter-leafings and task dependencies are preserved. For this, the
state space of the entire system architecture is constructed. Based on the state space response times are determined, and safety properties can be verified. As this approach is not scalable, we present abstraction techniques in order to deal with the large state space and realize a compositional analysis. The considered system class is a system architecture with a set of interconnected processing units (ECU). A set of tasks is allocated to each ECU. Tasks can have execution dependencies. For each ECU, the scheduling policy is given. The main focus will be on fixed priority scheduling (FPS) with preemption.

Next, we present in Section II and III the foundation of our approach. In Section IV, we present our state-based analysis technique. Abstraction techniques realizing a compositional approach. In Section V and finally we give a conclusion.

II. Timed Automata: Syntax and Semantics

Timed automata are finite automata extended with a finite set of real-valued variables called clocks. Timed automata were introduced by Alur and Dill in [1] in order to define a modeling concept for real-time systems. Here, we define syntax and semantics of timed automata as employed by Uppaal. Uppaal adapts timed safety automata introduced in [9]. In such automata, progress is enforced by means of local invariants. States (or locations) may be associated with a timing constraint defining upper bounds on clock variables.

In the sense of the classical Büchi Automata [1] all runs of a safety automaton are considered to be accepting.

Let \( C \) be a set of clocks. A clock constraint is defined by the syntax \( \varphi ::= c_1 \sim t \mid c_1 - c_2 \sim t \mid \varphi \land \varphi \), where \( c_1, c_2 \in C \), \( t \in \mathbb{Q}_{\geq 0} \), and \( \sim \in \{=, <, =, >, \geq \} \). The set of all clock constraints over the set of clocks \( C \) is denoted by \( \Phi(C) \). Valuation of a set of clocks \( C \) is a function \( \nu : C \rightarrow \mathbb{R}_+ \) assigning each clock in \( C \) a non-negative real number. We denote \( \nu \models \varphi \) the fact that a clock constraint \( \varphi \) evaluates to true under the clock valuation \( \nu \). We use \( 0_C \) to denote the clock valuation \( \{ c \mapsto 0 \mid c \in C \} \), abbreviate the time shift by \( \nu + d ::= \nu(c) + d \) for all \( c \in C \), and clock resets by \( \nu[\varnothing \mapsto 0] \) with \( \nu[\varnothing \mapsto 0](c) = 0 \) if \( c \in g \), and \( \nu[\varnothing \mapsto 0] = \nu(c) \) else for \( \varnothing \subseteq C \).

Definition 1 (Timed Automaton)

A Timed Automaton (TA) is a tuple \( A = (L, \mathbb{N}, \Sigma, C, R, I) \) where

- \( L \) is a finite, non-empty set of locations, and \( \mathbb{N} \subseteq \mathbb{N} \) is the initial location,
- \( \Sigma \) is a finite alphabet of channels, inducing the action set \( \Sigma^a = \{ a \mid a \in \Sigma \} \cup \{ \tau \mid a \in \Sigma \} \cup \{ \tau \} \), where \( \tau \) denotes internal actions,
- \( C \) is a finite set of clocks,
- \( R \subseteq L \times \Sigma \times K(C) \times \Sigma^C \times L \) is a set of transitions. A tuple \( r = (l, \sigma, \varphi, g, l') \) represents a transition from location \( l \) to location \( l' \) annotated with the action \( \sigma \), constraint \( \varphi \), and a set \( g \) of clocks which are reset,
- \( I : L \rightarrow \Phi(C) \) is a mapping which assigns an invariant to each location.

In the following, we will define timed transition systems which give semantics of timed automata.

Definition 2 (Timed Transition System) Let \( A \) be a timed automaton. The semantics of \( A \) is defined in terms of a timed transition system \( T(A) = (\mathit{Conf}, \mathit{Conf}^0, \rightarrow) \), where

- \( \mathit{Conf} = \{ (l, \nu) \mid l \in L \land \nu \models I(l) \} \) is the set of configurations, and \( \mathit{Conf}^0 = \{ (l', 0_C) \} \), where \( 0_C \) is the initial location and \( 0_C \) is the initial clock valuation.
- \( \rightarrow \subseteq \mathit{Conf} \times (\Sigma \times \mathbb{R}_+ \times \mathit{Conf}) \times \mathit{Conf} \) is the transition relation. A transition \( (l, \nu, \lambda, (l', \nu'), \phi) \), also denoted by \( (l, \nu) \rightarrow (l', \nu', \phi) \), has one of the following types:
  - A flow transition \( (l, \nu) \rightarrow (l, \nu + t) \) with \( t \in \mathbb{R}_+ \) can occur if \( \nu + t \models I(l) \). 
  - A discrete transition \( (l, \nu) \rightarrow (l', \nu') \) with \( \lambda \in \Sigma \) can occur if \( (l, \lambda, \varphi, 0, l') \in R \), such that \( \nu \models \varphi \), \( \nu' = \nu[\varnothing \mapsto 0] \) and \( \nu' \models I(l) \).

As the set of configurations is infinite, [1] gives a finite representation which is called region graph. In [3] a more efficient data structure called zone graph was presented. Basically, a zone represents the maximal set of clock valuations satisfying a corresponding clock constraint. Let \( g \in \Phi(C) \) be a clock constraint, the induced set of clock valuations \( D_g = \{ \nu \mid \nu \models g \} \) is called a clock zone. Let \( D^\uparrow = \{ \nu + d \mid \nu \in D \land d \in \mathbb{R}_+ \} \) and \( D[\varnothing \rightarrow 0] = \{ \nu[\varnothing \mapsto 0] \mid \nu \in D \} \). The finite representation of a timed automaton is given in terms of a symbolic transition system.

Definition 3 (Symbolic Transition System) Let \( A \) be a timed automaton. The symbolic transition system of \( A \) is a tuple \( \mathit{SST}(A) = (S, S_0, \rightarrow) \) where

- \( S = \{ (l, D_{1(l)}) \mid l \in L \} \) is the symbolic state set, and \( S_0 = \{ (l_0, D_{1(l_0)}) \} \) is the initial state,
- \( \rightarrow \subseteq S \times S \) is the symbolic transition relation with
  - \( (l, D) \rightarrow (l', D'[\varnothing \rightarrow 0] \cap D_{1(l')}) \) if \( (l, \sigma, g, l') \in R \).

Note, that for the general case some so called normalization operations on zones are necessary. If we build the symbolic transition system for an automaton containing clocks without a ceiling, i.e. some maximal reachable upper bound, it will lead to infinite sets of symbolic states. Nevertheless, for our cases the above definition will be sufficient as we always will have ceilings for all clocks. Please refer to [3] for more details on zone normalization operations.

Zones can be effectively represented by so called Difference Bound Matrices (DBM) [6], [2]. DBMs determine bounds on the difference between a set of clocks. Construction of DBMs can be found in [3].

III. Task Model and System Architecture

In this work we consider system architectures consisting of sets of processing units (ECU) which are directly interconnected. This is not a general restriction as analysis for CPU and bus scheduling is very similar. However, in future work we will consider communication resources (buses) explicitly. A set of tasks is allocated to each ECU. A task is
a tuple $t = (b_{ct}, w_{ct}, d, p_r)$, where $b_{ct}, w_{ct} \in \mathbb{N}_{\geq 0}$ are the best and worst case execution times with respect to the allocated ECU with $b_{ct} \leq w_{ct}$, and $p_r \in \mathbb{N}_{\geq 0}$ is the priority of the task. We will refer to the elements of a tasks by indexing, e.g. $b_{ct}t$ for task $t$. The set of all tasks is called $T$.

Independent tasks are triggered by events of a corresponding event stream (ES). An event stream $ES = (p, j, o)$ is characterized by a period $p$, a jitter $j$ and an initial offset $o$ with $p, j, o \in \mathbb{N}_{\geq 0}$. Such streams can be characterized by upper and lower occurrence curves as introduced in the real-time calculus [13]. In this work we will restrict to event streams where $j_t < p_t$ for all $t \in T$. Also more general event streams like bursts would be possible. Event streams and corresponding automata were presented in [8]. The timed automaton characterization of an event stream is depicted in Figure 1(b).

Figure 2 shows that task dependencies are captured by connecting corresponding tasks. If a task is directly connected to an event source it has no dependencies to other tasks.

Each ECU in a system is modeled by a tuple of elements. A mapping $\mathcal{T}$ determines the set of tasks that are allocated to the ECU. For each ECU, a scheduling policy is given. The main focus will be on fixed priority scheduling where preemption may lead to complex inter-leavings of task executions. Three additional functions provide dynamic bookkeeping needed to perform FPS analysis:

**Definition 4** An ECU is a tuple $\text{ecu} = (\mathcal{E}, \text{Sch}, \mathcal{R}, \mathcal{S}, \mathcal{A})$ where

- $\mathcal{T} : T \rightarrow B$ determines which tasks are allocated to a

considered ECU.
- $\text{Sch}$ determines the scheduling policy.
- A ready map $\mathcal{R} : T \rightarrow B$ determining tasks, which are released but to which no computation time has been allocated up to now.
- A start delay map $\mathcal{S} : T \rightarrow [t_1, t_2]$ with $t_1, t_2 \in \mathbb{N}_{\geq 0}$ which determines the delay interval for a task getting from status released to run.
- An active task map $\mathcal{A} : T \rightarrow [t_1, t_2]$ with $t_1, t_2 \in \mathbb{N}_{\geq 0}$ which determines the interruption times of tasks. This map is ordered and the first element determines the currently running task.

**IV. STATE-BASED ANALYSIS TECHNIQUE**

Our goal is to determine for each task corresponding response times, and whether all task deadlines and end-to-end deadlines are satisfied. The approach is based on explicitly constructing the symbolic transition system of the considered system. For this, we will first illustrate the construction for a single ECU, and then extend this approach for a set of interconnected ECUs. For the sake of readability, we will illustrate our techniques based on event streams without offsets and jitter. All algorithms and concepts can be extended to the more general event streams defined in Section III.

The symbolic transition system of an ECU has two locations for each task, one which indicates that this task is not released so far and another one indicating that this task
was previously released. We need this in order to capture the initial non-determinism of the input event streams of independent tasks.

The state set of an ECU mainly consists of clocks that capture the periodic behaviour of the allocated periodical tasks. For each independent task $\tau$ two types of clocks are needed. The first clock traces the periodical activation $c_p(\tau)$ of that task. Secondly, we have to trace the time frame from releasing the task up to the finish of computation. This kind of clock is called $c_{active}(\tau)$ in the following. In order to capture overlapping task activations, i.e., where multiple task instances $t_i$ of task $\tau$ may be active at the same time, multiple clocks $c_{active}(t_i)$ exist, one for each task instance. We need multiple clocks as we rely on using simple clocks in order to realize our scheduling analysis with preemption, i.e. we cannot change the derivative of a clock. Otherwise, we would have so called stopwatch automata where a stopwatch is used to track the allocated execution times of tasks. For this class of automata the reachability problem is known to be undecidable [4].

Our approach to trace the allocated execution times appropriately with clocks is similar to the approach illustrated in [7]: We reset $c_{active}(t_i)$ when $t_i$ is released. If the task is not interrupted by other tasks, we can determine when this task will finish its computation, namely when $bcet_{t_i} \leq c_{active}(t_i) \leq wcet_{t_i}$. When preemptions occur, this constraint is changed to $bcrt_{t_i} \leq c_{active}(t_i) \leq wcrt_{t_i}$, i.e. the accumulated response times.

As we need to use one separate clock per task instance, we need to know a priori the maximal number of possible parallel activations of one task. This number always can be derived from the task period and its deadline.

In our algorithm we use the DBM representation of zones. We will use the standard notations for matrices, i.e. $D_{i,j}$ is the element in the $i^{th}$ row and $j^{th}$ column. Each element in a DBM determines a clock difference, i.e. $D_{i,j} = n$ represents the inequation $c_i - c_j \leq n$ where $c_i,c_j \in C$. Note, that index 0 is always allocated to the reference clock. So, $D_{j,0}$ gives the upper bound of clock $c_j$, and $D_{0,j}$ the lower bound. For the sake of readability, we will use tasks as indexes, i.e. for task $\tau$ we refer with $D_{\tau}$ to $c_p(\tau)$ and for a task instance $t$ with $D_t$ we refer to $c_{active}(t)$. For an interval $i$ we use the functions $i.lb()$, $i.ub()$ to access the lower and upper bound of $i$.

Our algorithm starts with the initial symbolic state $\langle \emptyset, D_{I(\emptyset)} \rangle$ and proceeds as follows: Each state which is generated is added in a list. This list determines the state set, for which the successors have to be computed. The successor computation algorithm is illustrated in Figure 3. First, according to Definition 3, we compute the reachable time successor $D^{Post}$. The time successor is defined by the following invariants: First we have invariants determined by the periodicities of the tasks, i.e. $c_p(\tau_i) \leq p_{\tau_i}$ for tasks $\tau_i$ allocated to the considered ECU. Further, if a task instance $t_i$ is running, we get the additional invariant $c_{active}(t_i) \leq wcet_{t_i} + interruptTimes_{t_i}.ub()$. This is illustrated in the right part of Figure 4. The interrupt times for a task are tracked in the active task map $\mathcal{A}$.

After determining $D^{Post}$, we make a case distinction: if there is no running task in the current state, we build the successor state by taking the discrete step which releases an instance of task $\tau$. If the computed successor does not exist yet, it is added to the graph by building an edge from the current state $(l, D)$ to the computed successor.

The discrete step operation is depicted in the left part of Figure 4. If $t_i$ is a newly released task instance (indicated by $term = false$), then the lower bound of its clock is set with respect to the event stream from which it is activated: if the task $\tau$ is released the first time, the activation can occur at $0 \leq c_p(\tau) \leq p_\tau$, else it occurs exactly at $p_\tau$. Note, that here we have only to handle the lower bounds of the clocks as the upper bounds are already handled by the invariant intersection operation. On the other hand, if we have a task termination, $c_p(\tau)$ is not touched but rather $c_{active}(t_i)$. If a task terminates, we accumulate its execution time to all interrupted tasks by incrementing their interrupt times in the active task map $\mathcal{A}$ of the corresponding ECU. This is analogous to classical scheduling analysis methods, where response times are computed by a fixed point equation propagating the interruptions of a task.

If we have a running task for a given state, we need to determine whether this task terminates. This is indicated in the code fragment of Figure 3 by calling the handlePossibleCases method. This method operates as follows:

1) If neither an activation of a new task instance $t$ is possible nor the termination of the current running task $t'$, i.e. $c_{active}(t').ub() < bcrt_{t'} \land c_p(t) \leq p_\tau$, then nothing is done.

2) If the running task $t$ cannot terminate, i.e. $c_{active}(t).ub() < bcrt_t$, but a new task instance $\tau$ can be released, then takeDiscreteStep$(\tau,false)$ is called. After this, we determine the next running task with respect to the considered scheduling policy, and update the maps accordingly like it is depicted in the code fragment of Figure 5. If we move the task $t$ from the ready map to the active map, we have to store the time frame from release to start of $t$ in order to correctly determine its allocated execution time. This time frame is given by clock $c_{active}(t)$ (or rather $D_t$ in the matrix) and is stored in the start delay map $S$ (see line 4 in Figure 5). At least the new state is added to the graph.

3) If the running task $t$ will terminate ($c_{active}(t).lb() = wcrt_t$), then takeDiscreteStep$(t,true)$ is called. Then analogous to case 2) we determine the next running task, update the ready map and add the new state. When adding the new state we annotate the edge with the response time of
the terminating task. At least, we are recursing computeSuccessor \((l', D'), \tau\).

If \(c_{\text{active}}(t) \cdot \text{ub}() \geq \text{bcrt}_t \land c_{\text{active}}(t) \cdot \text{lb}() \leq \text{wcr}_t\) then both 2) and 3) are executed in this order. Note, that when a task terminates (case 3) its response time is simply added to the created edge. When we want to determine the response times of tasks we simply iterate through the constructed graph and search for the minimum and maximum values. With this, we can easily determine violations of deadlines.

Extending the technique illustrated above to a set of interconnected ECUs is canonical: For dependent tasks, we only need the class of clocks tracking the activation time \((c_{\text{active}})\). Task releases of dependent tasks are determined by finishing events of the tasks from which they are depending. The only thing we further need to do is to reset the clocks each time when they are not used. Further, we need data structures handling the set of maps and task allocations.

V. COMPOSITIONAL ANALYSIS THROUGH ABSTRACTION TECHNIQUES

Unfortunately, the presented approach is not scalable as we get many clocks leading to large state spaces. It is well known that complexity rises exponential with the set of used clocks. We present abstraction techniques that reduce the state space due to the large amount of clocks. The first technique is not new in general, but we show how it can be adapted to our technique. The aim here is to reduce the state space due to the large amount of clocks. We present abstraction techniques that reduce the state space.

A abstraction function generally maps a state set to another state set, i.e. is a function \(\alpha : S \rightarrow S'\) where \(S' \subseteq S\). Two concrete abstraction functions \(\alpha_{ES}, \alpha_{dur}\) will be presented in the following subsections. First, we need some useful definitions.

Let \(C' \subseteq C\) be a subset of clocks. For a clock constraint \(g\) over \(C\) let \(g_{|C'}\) be the constraint where all propositions containing clocks of the set \(C \setminus C'\) are removed. For example, consider the sets \(C = \{c_1, c_2\}, C' = \{c_2\}\) and the constraint \(g = c_1 \land c_2 \leq 3 \land c_1 \leq 5 \land c_2 \leq 1\). Then \(g_{|C'} = c_2 \leq 1\). For a zone \(D = \{\nu \mid \nu \models g\}\) defined over \(C\) we define the zone projection operation \(D_{|C'} = \{\nu \mid \nu \models g_{|C'}\}\) accordingly. Note that we have \(D_g \subseteq D_{C'}\).

Let \(L\) be a set of tuples of locations \(l_1, ... l_n\). These locations are indexed by the set \(I = \{1, ... n\}\). Let \(I' \subseteq I\) be a subset of these indexes. Then \(L_{|I'}\) is a set of tuples of locations where locations with indexes not in \(I'\) are left out.

Let \(L = \{\{t_0, t_{10}...0\}, \{t_{01}0...0\}, ..., \{t_{11}1...1\}\}\) be the set of locations of the symbolic transition system, where \(t_{00}...0\) indicates that an instance of task \(\tau_1\) has already been released and \(l_{11}...1\) indicates all tasks have already been released. Then \(L_{\{\tau_1, ..., \tau_{i-1}, \tau_{i+1}, ..., \tau_n\}\}\) contains only the locations of all tasks but \(\tau_i\).

A. Computing Event Streams

Let \(G\) be a symbolic transition system for an ECU containing only independent tasks. The abstraction function leading to output event streams is defined as follows:

\[
\alpha_{ES} : \{\{l, D\} \mid l \in L\} \rightarrow \{\{l, D_{|C}\} \mid l \in L_{|\{\tau\}}\},
\]

where \(C_\tau = \{c_p(\tau), c_{\text{active}}(t_1), ..., c_{\text{active}}(t_n)\}, t_i\) are instances of task \(\tau\) and \(n\) is the number of maximal overlapping activations.

The result of applying this abstraction operation for a task with \(n = 1\) is illustrated in Figure 6, which depicts the ranges of the clock given in the corresponding DBMs. One can see that this graph is the symbolic state set of the standard event streams defined in Section III where jitter and offset are omitted as stated before.

With this, we can directly apply a compositional analysis technique, where all computed output event streams of all ECUs with independent tasks are used to compute the state set of ECUs with dependent tasks.

Indeed with this abstraction technique all interleavings of tasks are lost. In fact, the same technique is applied for many classical compositional scheduling analysis approaches.

B. Duration Clock Abstraction

As we have seen in Section IV we need for all possible overlapping activations of a task one separate clock that traces the time frame since the instance has been released. The idea now is to abstract from the exact response times of all single instances to just the termination times of a task type. For this, we will abstract from the set of clocks of a task to just one single clock, which tracks the distances of two subsequent terminations of this task. With this, a reduction of the number of clocks is realized while keeping the interferences of other tasks. The idea of this technique is illustrated in Figure 7, where two successive activations of task \(\tau\) occur. The clocks for task instances \(t_1, t_2\) exactly trace the response times of each instance. When abstracting
from these we get clock \( d \) only tracing the termination times of the task instances.

Technically, we extend the set of clocks of a resource by further clocks \( DC \) that track the distances of subsequent task terminations. In this set, exactly one clock for each task is allocated, i.e., \( c_d(\tau) \) for task \( \tau \). The clock \( c_d(\tau) \) is rested, when a task instance of \( \tau \) terminates. These clocks will be called duration clocks.

We build the state space of the considered symbolic transition system with the above extension. After that, we determine the valuations of the duration clocks of each task. With this, we get a set of extended in-equations over \( C \cup DC \). Note that the extended constraints are only derived constraints that do not change the original zone sets, i.e., for all zones \( D \) of constraints \( g \in \Phi(C) \) and \( D^+ \) of extended constraints \( g^+ \in \Phi(C \cup DC) \) it holds that \( D^+_C = D \). Hence, when we map the extended constraints to constraints of the original clock set, we get the same set of clock valuations. With this, we define our abstraction function as follows:

\[
\alpha_{dur} : \{l, D^+ \} \mid l \in L \to \{l, D^+_C \mid l \in L\}.
\]

When no overlapping activations of a task occur, the duration clocks do not lead to abstractions. For this, consider Figure 8. The derived constraints are depicted in the locations of the graph.

VI. CONCLUSION AND FUTURE WORK

We presented a scheduling analysis technique for fixed priority scheduling with preemption that is based on model-checking techniques known for the well-established formalism of timed automata. While FPS with preemption is known to mark a boundary of decidability for this class of systems, exploitation of results from analytical scheduling analysis enables a suitable representation for the considered transition systems. We presented abstraction techniques leading to state space reduction due to large amount of clocks.

Future work includes addition of other scheduling policies like TDMA, the refinement of abstraction techniques, and analysis of their impact with respect to the resulting state space reduction and tightness of analysis results.

REFERENCES


