Real-Time Task Models

Yue Tang, Nan Guan and Wang Yi

Abstract In safe-critical real-time systems, it is required that the timing constraints must be respected under any circumstance at runtime. To provide such guarantees, the designers need to describe the system behaviors with formal models, based on which timing correctness of the system can be proved. There have been many different real-time task models developed over the years describing different timing behaviors of real-time systems. This article briefly reviews some of the representative real-time task models, focusing on the expressiveness aspect. In particular, this article will cover the basic periodic/sporadic task models with extensions of bursts and jitters, the graph-based real-time task models which can model different possible workload release patterns in a compact model, the parallel real-time task models that can describe the internal parallel workload structures and the arrival curves in Real-Time Calculus as a general workload representation.

Introduction

Real-time systems are computing systems in which the correctness of system behaviors depends on not only the logical computation results, but also the physical time when these results are produced (Stankovic, 1992) . Many real-time systems are safe-critical, so timing constraints must be respected under any circumstance at runtime. To provide such guarantees, the designers need to describe the system behavior with formal models, based on which timing correctness of the system can be proved. There are three major aspects in real-time system modeling: workload,

Yue Tang
The Hong Kong Polytechnic University, e-mail: csyuetang@comp.polyu.edu.hk

Nan Guan
The Hong Kong Polytechnic University, e-mail: nan.guan@polyu.edu.hk

Wang Yi
Uppsala University, e-mail: yi@it.uu.se

resource and scheduling policy. This article focuses on the workload aspect of real-time system modeling.

Real-time systems are often implemented by a number of concurrent tasks sharing the hardware resource. The workload of a real-time system is characterized by *real-time task models*, which typically have the following three features:

- **Recurrence**. A real-time task is recurrently activated for infinitely many times.
- **Deadlines**. The workload of a real-time task incurred by each activation is required to be finished before a certain time point.
- **Multitasking**. The system consists of multiple concurrent real-time tasks.

The combination of the above factors makes the real-time task models unique comparing with workload models in other research areas, e.g., operational research. The deadline feature is closely related to the makespan minimization problem and also studied in many other research areas. However, due to the (infinite) recurrence feature of real-time tasks, it is a difficult problem to analyze whether the deadlines are always met in general, which cannot be solved by standard constraint-solving and optimization techniques with finite state space.

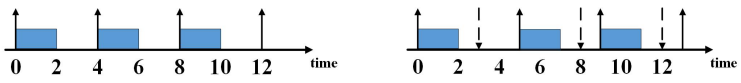
Over the years, a large number of *real-time task models* have been developed to model different workload release patterns. Some model is as simple as a periodic pattern characterized by two parameters, while some of them are as expressive as Turing machine. In general, the simpler the task model is, the easier to analyze it. Therefore, the development of real-time task models usually takes the underlying analysis techniques into consideration. This article will briefly review some representative real-time task models, and only focus on their expressiveness aspect. Due to the page limit, we can only cover a small part of real-time task models in literatures. The remainder of this article is organized as follows. Section 2 introduces the basic periodic/sporadic task models with extensions of bursts and jitters. Section 3 introduces the graph-based real-time task models. Section 4 introduces the parallel real-time task models. Section 5 introduces arrival curves in Real-Time Calculus. Section 6 briefly summarizes the article.

Periodic/Sporadic Real-Time Task Models

Basic Periodic/Sporadic Task Models

The basic periodic real-time task model proposed in Liu and Layland (1973) captures the fundamental periodically recurring behavior of many real-time computing systems (Leung and Merrill, 1980; Leung and Whitehead, 1982). A periodic task set τ consists of a set of independent periodic tasks $\{\tau_1, \tau_2, \cdots\}$. A periodic task τ_i is characterized with two parameters, the worst-case execution time (WCET) C_i and the period T_i. A periodic task generates an infinite sequence of *jobs* (or called *task instances*). The first job of a task may be released at any time, and each subsequent job is released with a fixed separation time T_i from its predecessor job. A job requires to execute for at most C_i time units. The execution must be finished before the release of the next job, i.e., task τ_i has a *relative deadline* equal to its period T_i and each job has an *absolute deadline* aligned with the release time of the next job. A job sequence of a periodic task τ_i with $C_i = 2$ and $T_i = 4$ is shown in Figure 1(a). A solid up arrow in the figure represents the release time of a job (also the absolute deadline of its predecessor job).

The sporadic task model generalized the Liu and Layland's periodic task model to allow more flexible release patterns and deadline constraints (Baruah, Mok, & Rosier, 1990; Lehoczky, Sha, & Strosnider, 1987; Mok, 1983; Sprunt et al., 1989). A *sporadic* task τ_i is characterized with three parameters, the worst-case execution time (WCET) C_i, the minimum inter-release separation T_i and the relative deadline D_i (Mok, 1983). The inter-release separation between two consecutive jobs generated by a sporadic task is allowed to be larger than T_i. Moreover, a sporadic task τ_i may have a relative deadline D_i different from its period T_i. We say a sporadic task τ_i has an *implicit deadline* if $D_i = T_i$, a *constrained deadline* if $D_i \leq T_i$ and an *arbitrary deadline* if there is no limitation on the relation between D_i and T_i. A job sequence generated by a sporadic task τ_i with $C_i = 2$, $T_i = 4$ and $D_i = 3$ is shown in Figure 1(b). A dashed down arrow represents the absolute deadline of its predecessor job. The first job is released at time 0, and the second job is released at time 5, with an inter-release separation larger than $T_i = 4$. The third job is released at its earliest possible release time 9, which is 4 time units after the last job release time 5. The dashed down arrows represent the absolute deadlines.



(a) A periodic task. (b) A sporadic task.

Fig. 1: Examples of a periodic task and a sporadic task.

Extensions with Jitter and Burst

The periodic real-time task model was extended by adding a jitter J_i for each task τ_i (Audsley et al., 1993), to allow the flexibility that the actual release time of a job may be at most J_i time units earlier or later than the exact start time of the period. Due to the jitter, two jobs may be released with a separation smaller than the period, which causes a local workload burst. Tindell et al. introduced another extension of periodic/sporadic tasks to model bursts (Tindell, Burns, & Wellings, 1994). Each task has two periods: the inner period and the outer period. The inner period denotes the minimum inter-release separation of events inside a burst, and the outer period denotes the minimum inter-release separation between different bursts. This burst model controls the maximum number of events that need to be processed in a certain time interval while guaranteeing a long-term event arrival rate.

A more general extension of periodic/sporadic tasks is the *PJD event model* (Richter et al., 2003), where each task τ_i is characterized by three parameters: period P_i, jitter J_i and minimum inter-arrival time D_i (notice here D_i does not represent the relative deadline). Although the original PJD event model only focuses on the release pattern of the workload but does not model the time to process each event, it is easy to extend the model by adding the WCET information. Figure 2(a) illustrates several release patterns allowed by the PJD event model.

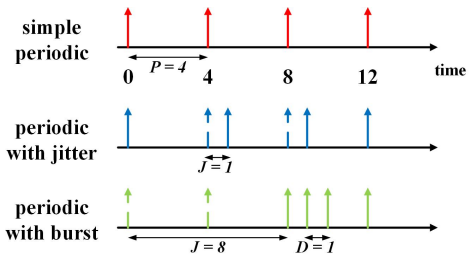


Fig. 2: Several release patterns of a PJD task.

Offset-based Task Models

In the *Offset-based task model* (Palencia and Harbour, 1998, 2003; Tindell, 1994), a task τ_i has a period T_i as in the periodic/sporadic task model and includes several subtasks $\{\tau_i^1, \cdots, \tau_i^x\}$. In each period, the release time of a subtask τ_i^k has an offset O_i^k relative to the start time of this period. The release of each subtask τ_i^k also allows a jitter J_i^k. Moreover, an *every* attribute e_i^k describes that a subtask τ_i^k can be released once in at most e_i^k periods of the task, and this subtask has an associated period $e_i^k T_i$.

Using the offset-based task can reduce input and output jitters in data transmission, for which an example was given by Tindell (1994). Suppose a task first does some computation and then transmits the output data. The time for the task to send the output data may have a large jitter, since the computation process may be influenced by many factors such as task preemptions. To reduce the output data jitter, we decompose the task into two subtasks A and B. A only performs the computation and the output data are temporally stored. B takes charge of transmitting the output data. We put A and B in an offset-based task, and let B have an offset large enough to cover the jitter of A. By giving task B a higher priority, we can guarantee that latter data can not be produced until the former data have been transmitted. Since task B only implements a simple operation, its jitter is relatively small, and the overall jitter of the output data emission is reduced.

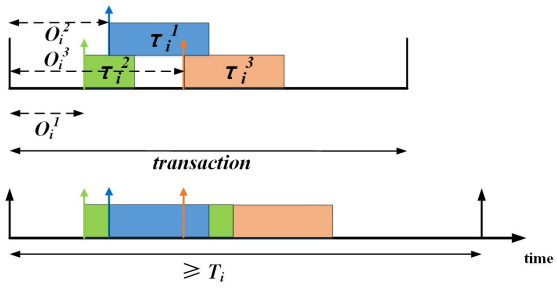


Fig. 3: An offset-based task and one possible execution pattern.

In Figure 3, a sporadic offset-based task τ_i with period T_i has 3 subtasks τ_i^1, τ_i^2, τ_i^3, among which τ_i^1 has the highest priority and τ_i^3 has the lowest priority. The offsets of τ_i^1, τ_i^2, τ_i^3 are O_i^1, O_i^2, O_i^3. The figure in below shows a possible execution pattern. At some time instant, τ_i^2 is released with an offset O_i^2 relative to the start of the current instance of τ_i and starts to execute. τ_i^1 is released with an offset more than O_i^1 and preempts the execution of τ_i^2. During the execution of τ_i^1, τ_i^3 is released with an offset O_i^3. Since τ_i^3 has the lowest priority, it has to wait until τ_i^1 and τ_i^2 both finish their execution. Since the task is sporadic, the next instance of task τ_i is released with a minimum inter-release separation T_i from its predecessor instance.

Graph-based Real-Time Task Models

The simple periodic/sporadic task models cannot describe the complex timing behaviors of many realistic real-time applications. This section introduces the extension of the periodic/sporadic task model using graph structures to represent more complex workload release patterns. The periodic/sporadic task model can also be viewed as a simple case of graph-based task model, which has only one node representing the release of workload and one self-loop edge representing the inter-release separation. Figure 4 summarizes the graph-based task models to be introduced in this section. In the following we introduce them in the order from more restricted to more expressive ones.

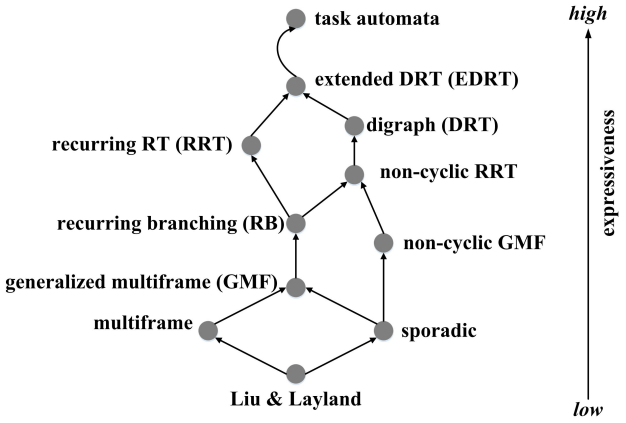


Fig. 4: A hierarchy of graph-based task models.

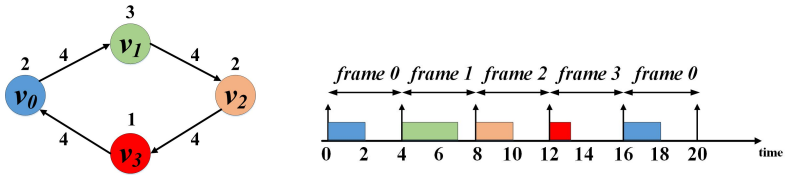
Multiframe Task Models

In many real-time systems the worst-case execution time of the job generated in one invocation of a task may be different from that of another invocation. As an example, Mok and Chen (1997) described an MPEG video codec that uses different types of video frames. Video frames arrive periodically, but frames of a certain type with large size (and thus large decoding complexity) arrive only once in several consecutive frames. The sporadic task model needs to account for this in the WCET of all jobs, which is a very pessimistic overapproximation. Many schedulable systems would fail standard schedulability tests for the sporadic task model.

To solve this problem, Mok and Chen (1997) introduced the Multiframe (MF) task model. A multiframe task τ_i of k frames is described as a pair (T_i, C_i) similar to the basic sporadic task model with implicit deadlines, except that $\mathsf{C}_i =$

$\{C_i^0, \cdots, C_i^{k-1}\}$ is a vector of different WCETs of k different frames. A multiframe task can be viewed as a cycle $G = \langle V, E \rangle$ as shown in Figure 5(a). A node $v \in V$ represents a frame labeled with $c(v)$, the WCET of frame v. Each edge is labeled with the minimum inter-release separation T_i between the two nodes u, v it connects, which is the same for all edges. Each node has an implicit deadline, i.e., the relative deadline of each node equals the minimum inter-release separation labeled on its outgoing edge.

The execution of an MF task starts with an arbitrary node, traverses along the edges, and iterates along the cycle repeatedly. A job sequence is denoted by $J = (J_0, J_1, \cdots)$ with job parameters $J_k = (r_k, c_k, v_k)$ of release time r_k, execution time c_k and job type v_k. An example of an MF task is shown in Figure 5(b). A job sequence of $\{(0,2,v_0),(4,3,v_1),(8,2,v_2),(12,1,v_3),(16,2,v_0),\cdots\}$ corresponds to a path $\{v_0, v_1, v_2, v_3, v_0, \cdots\}$ in the cyclic task graph.



(a) A multiframe task. (b) Possible release pattern.

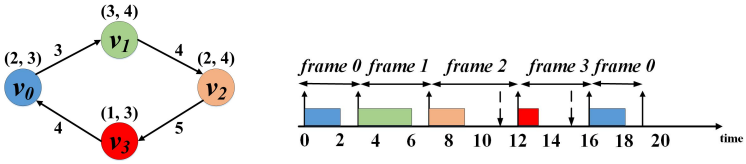
Fig. 5: A multiframe task and one possible release pattern of it.

The Generalized Multiframe (GMF) task model (Baruah et al., 1999; Takada and Sakamura, 1997) extends MF to allow nodes to have not only different WCETs, but also different relative deadlines and allow different minimum inter-release separations on different edges. A GMF task τ_i with k frames is characterized by three vectors $(\mathsf{P}_i, \mathsf{E}_i, \mathsf{D}_i)$. $\mathsf{P}_i = (P_i^0, \cdots, P_i^{k-1})$ describes the minimum inter-release separations. P_i^j represents the minimum separation between the release time of an instance of frame j and its successor frame. $\mathsf{C}_i = (C_i^0, \cdots, C_i^{k-1})$ describes the WCETs of the frames and $\mathsf{D}_i = (D_i^0, \cdots, D_i^{k-1})$ describes their relative deadlines.

A GMF task can also be represented by a cycle $G = \langle V, E \rangle$. Each node $v \in V$ is labeled with $(c(v), d(v))$, where $c(v)$ is the WCET of frame v and $d(v)$ is the relative deadline. Each edge (u, v) is marked with its own minimum inter-release separation $p(u, v)$. The relative deadline $d(v)$ of each node v in general can be different from its minimum inter-release separation of the outgoing edge (similar to an arbitrary-deadline sporadic task).

An example of a GMF task with 4 different frames is shown in Figure 6(a). Suppose the task starts execution with *frame 0* at time 0. A job sequence (defined in the same way as in an MF task) $\{(0,2,v_0),(3,3,v_1),(7,2,v_2),(12,1,v_3),(16,2,v_0),\cdots\}$ is shown in Figure 6(b).

Both MF and GMF follow a cyclic execution pattern. The *non-cyclic GMF* task model (Baruah, 2010a; Moyo et al., 2010) allows to model the execution of different



(a) A generalized multi-frame task.

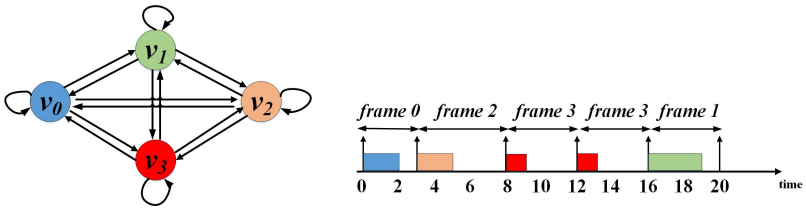
(b) Possible release pattern.

Fig. 6: A generalized multiframe task and one possible release pattern of it.

frames not following a cyclic pattern. The same as GMF tasks, a non-cyclic GMF task τ_i is also represented by three vectors $(\mathsf{P}_i, \mathsf{C}_i, \mathsf{D}_i)$ but with different semantics. To define the semantics formally, let $\phi : \mathbf{N} \to \{0, \cdots, k-1\}$ be a function choosing frame $\phi(j)$ for the j^{th} job of a job sequence. With ϕ, each job $J_j = (r_j, c_j, v_j)$ in a job sequence $(J_0, J_1, \cdots)$ generated by a non-cyclic GMF task needs to correspond to frame $\phi(j)$ and the corresponding values in all three vectors:

$$r_{j+1} \geq r_j + P_{\phi(j)} \ \wedge \ c_j \leq E_{\phi(j)}$$

A non-cyclic GMF task can also be modeled as a complete graph. Figure 7(a) shows an example of non-cyclic GMF and Figure 7(b) shows a possible job sequence $\{(0,2,v_0),(3,2,v_2),(8,1,v_3),(12,1,v_3),(16,3,v_1),\cdots\}$.



(a) A non-cyclic GMF task.

(b) Possible release pattern.

Fig. 7: A non-cyclic GMF task and one possible release pattern of it.

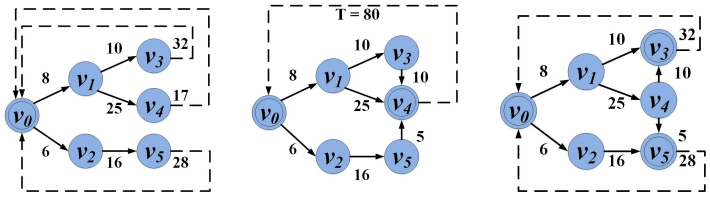
Recurring Branching Task Models

The code of many real-time systems may include branches that influence the workload release patterns. In general, the workload released by different branches may be incomparable. Thus, we need to model the branching semantics explicitly in the task model. To this end, Baruah proposed the Recurring Branching Task Model in Baruah (1998a). A *Recurring Branching* (RB) task (Anand et al., 2008) is repre-

sented by a directed tree structure $G = (V, E)$. Similar to the graph representation of multiframe task models, each node $v \in V$ is labeled with with $(c(v), d(v))$, specifying the WCET and relative deadline of the node. A node may have multiple outgoing edges describing possible branches. There are two types of edges: edges with sink nodes labeled with the minimum inter-release separation $p(u, v)$ between two jobs u, v, and edges without sink nodes labeled with the minimum inter-release time $p(v_l, v_{root})$ between its source node (a leaf node in tree structure) v_l and the root node v_{root} .

The execution starts at an arbitrary node and traverses the graph along the edges. When a node has more than one outgoing edge, only one edge is chosen for execution at one time. After a leaf node is executed, the execution starts again from the root node and the new recursion may traverse different branches from last recursion. Since the self-loop structure is not allowed, each node can only be visited once in each iteration. A general period T is defined as the minimum inter-release separation between two releases of the root node. All paths from the root node to a leaf node have the same sum of inter-release separation times and this sum is the general period T.

Figure 8(a) shows an example of an RB task. Since WCETs and relative deadlines of nodes are the same as former task models, they are omitted in the figure and we focus on the semantics of the inter-release separations of the example. There exists three branches, (v_0, v_1, v_3), (v_0, v_1, v_4), and (v_0, v_2, v_5), from the root to the leafs, with the same general period 50.



(a) An RB task. (b) An RRT task. (c) A non-cyclic RRT task.

Fig. 8: The RB, RRT and non-cyclic RRT tasks.

In typical branching code, the control flows joined again after branches are completed. Thus, it is redundant to have one leaf node for every branch. In the *recurring real-time task (RRT)* model (Baruah, 2003; Chakraborty, Erlebach, & Thiele, 2001) proposed by Baruah (1998b), this redundancy was removed by combining all leaf nodes into one sink node, and using a *directed acyclic graph (DAG)* instead of a tree. Different from RB tasks, an RRT task does not have a minimum inter-release separation constraint from the sink node to the source node, but specifies a minimum separation between two consecutive release times of the root node. Figure 8(b) shows an example of an RRT task. v_0 is the root node, and v_4 is the sink node.

There is no constraint on the minimum inter-release separation from v_4 to v_0, but two consecutive releases of v_0 must be separated by at least 80 time units.

Baruah (2010b) generalized the RB model into *non-cyclic RRT* model, where the constraint of one single sink vertex was removed. A non-cyclic RRT task is a DAG with vertex and edge labels as before and a unique root vertex v_{root}. For every sink vertex v, there is a value $p(v, v_{root})$. A non-cyclic RRT task does not have a general period parameter. In other words, the time intervals between two consecutive visits of the root node may not be same when different sink nodes are visited.

Figure 8(c) shows an example of non-cyclic RRT task. v_0 is the root node, and v_3, v_5 are sink nodes. There is minimum inter-release separation from v_3, v_5 to v_0, but no general period for the whole task. The non-cyclic GMF and non-cyclic RRT share the similarity that they are both recurring but do not have a fixed general period. Non-cyclic GMF is a special case of non-cyclic RRT, and a non-cyclic GMF task can be transformed to a non-cyclic RRT task (Stiggie, 2014).

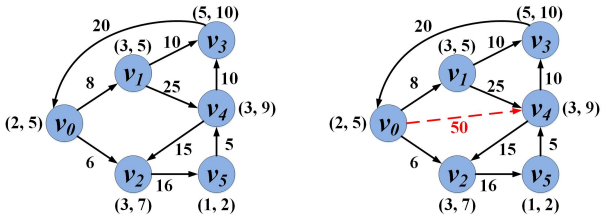
Digraph Real-Time Task Models

The graph-based task model can be generalized to any directed graph. Stigge et al. (2011a) introduced the *Digraph Real-Time (DRT)* task model (Mohaqeqi et al., 2016; Stigge and Yi, 2012, 2013, 2014). A DRT task is described by a directed graph with edge and vertex labels as before. There are no further restrictions, and any directed graph can be used to describe a task. Using any graph allows modeling local loops which are not possible in any model presented above. Even in the non-cyclic RRT model, all cycles in that model have to pass through the source vertex. An example of DRT is shown in Figure 9(a). The execution could start at any node and there exists a self-loop for v_0. A job sequence of $\{(0,2,v_0),(10,3,v_1),(20,5,v_3),(50,2,v_0),\cdots\}$ corresponds to the path $\{v_0,v_1,v_3,v_0,\cdots\}$ in the graph.

DRT is further extended to the *extended DRT (EDRT)* task model (Stigge et al. , 2011b) by allowing specifying the minimum inter-arrival time between two nodes that are not connected directly. In addition to a graph as in the DRT model, an EDRT task also includes a set of *global inter-release separation constraints*. Each constraint $(from_k, to_k, \gamma_k)$ expresses that between the visits of nodes $from_k$ and to_k, at least γ_k time units must pass. An example of EDRT is shown in Figure 9(b). The dashed edge from v_0 to v_4 shows that at least 50 time units have to pass between the release of these two nodes.

Task Automata

Task automata (Fersman, Pettersson, & Yi, 2002) is a very expressive model based on graph structures, extending the well-known timed automata model (Bengtsson



(a) DRT. (b) Extended DRT.

Fig. 9: A DRT task and its extension to EDRT.

and Yi, 2003) to model task release patterns. Timed automata are finite automata extended with real-valued clocks to specify timing constraints as enabling conditions, i.e., guards on transitions. The essential idea of task automata is to use the timed language of an automaton to describe task release patterns.

A task automaton can also be represented by a graph $G = (V, E)$. Similar to DRT, each node $v \in V$ represents the release of a type of job, with WCET $c(v)$ and relative deadline $d(v)$. A task automata has an initial node which is the start of each execution. Edges still represent the order of execution, but the labels on edges are more expressive. An edge (u, v) is labeled with a *guard* $g(u, v)$ which is a boolean combination of clock comparisons of form $x \triangle C$ where C is a nature number and $\triangle \in \{\leq, <, \geq, >, =\}$. An edge may be labeled with a clock reset $r(u, v)$ which resets the clock to be 0 when this edge is taken. Since the value of clocks is an increasing real value which represents that time passes, guards and resets can be used to constrain timing behaviors on generated job sequences. Apart from the timing constraints, the edges may also be labeled by external events representing the trigger of the transition.

An example of task automata is shown in Figure 10. v_0 is the initial node. Event a triggers the release of v_1, which will execute for at most 2 time units. After 10 time units passed, the transition from v_1 to v_2 is enabled, and v_2 can be released at any time after that by taking the transition. When the clock y counts more than (equal to) 50 time units, the transition of edge v_2 to v_3 is taken, and clock x is reset to 0. The transition from v_3 to v_1 needs to be triggered by event b. So if event b arrives before clock x counts 5 time units, the transition from v_3 to v_1 is taken. If event b arrives late, then v_3 will repeat its computation without stop.

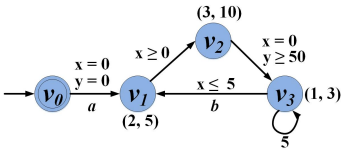


Fig. 10: An example of task automata

Parallel Real-Time Task Models

When real-time systems are executed on parallel processing architectures (e.g., multiprocessors and multi-cores), it is meaningful to explore the parallelism of the workload to better utilize the resource. Traditional real-time systems assume full independence among tasks. However, this is not the case when the workload consists of several processes communicating and cooperating with each other. As a result, task models that can describe fine-grained synchronizations among tasks (Feitelson and Rudolph, 1992) are requested.In this section, we introduce several parallel real-time task models.

Gang Task Model

The *gang task model* (Berten, Courbin , & Goossens, 2011; Dong and Liu, 2017) describes the simple parallel execution behavior where the task executes on several processing units simultaneously. A gang task τ_i is characterized by (C_i, m_i, D_i, T_i) where D_i and T_i are relative deadlines and minimum inter-release separations as before. The task contains m_i parallel threads, which must execute simultaneously. C_i denotes the WCET of each thread, and thus the total workload of the task is $C_i \times m_i$.

Figure 11 (b) shows an example of a task set with 3 sporadic gang tasks with implicit deadlines executing on 4 processors. Task τ_1 is characterized by $C_1 = 2$, $m_1 = 2$, $D_1 = T_1 = 10$, task τ_2 is characterized by $C_2 = 2$, $m_2 = 2$, $D_2 = T_2 = 5$, and task τ_3 is characterized by $C_3 = 3$, $m_3 = 3$, $D_3 = T_3 = 5$. We assume τ_1 has the highest priority, and τ_3 has the lowest priority. At time 0, tasks τ_1, τ_2, τ_3 are released simultaneously, and τ_1, τ_2 start execution first due to their higher priority. At time 2, τ_1 and τ_2 finish execution and τ_3 starts execution. At time 5, τ_2 is released again, and it starts execution. τ_2 only occupies 2 processors, but τ_3 cannot execute from time 5 since all its threads must execute simultaneously and only one idle processor is not enough. At time 7, task τ_2 finishes and task τ_3 starts execution. Finally task τ_3 completes its execution at time 10.

Parallel Synchronous Task Models

The *fork-join task model* introduced in Lakshmanan, Kato, & Rajkumar (2010) describes structures with interleaving sequential and parallel segments which are common in parallel programs such as OpenMP (OpenMP, 2018; Sun et al., 2017; Wang et al. , 2017). A segment is a continuous section of computation operations. The execution of a fork/join task starts with a sequential segment, and forks into several parallel sections in a parallel segment, then joins together at the end of segment. This

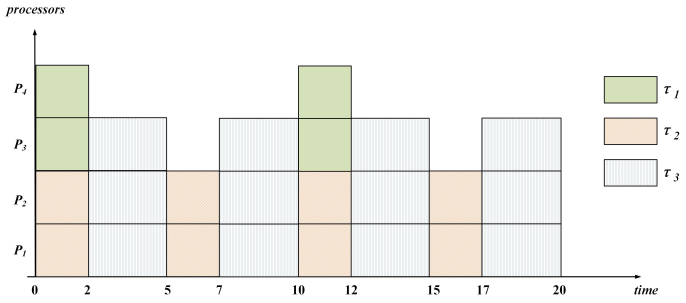
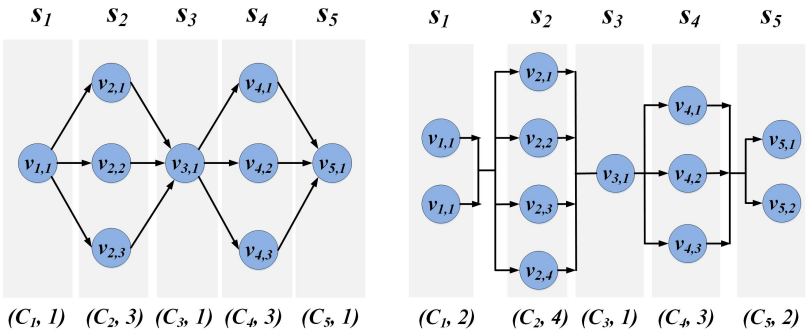


Fig. 11: One possible execution pattern of a sporadic gang task.

pattern repeats for multiple times in a task, and the number of threads in different parallel segments is same.

A fork/join task τ_i is also characterized by $(\mathsf{C}_i, m_i, D_i, T_i)$, where D_i is the relative deadline and T_i is the minimum inter-release separation, which are similar with before. m_i is the parallelism, i.e., number of threads, of each parallel segment. All parallel segments share the same m_i. $C_i = \{C_i^1, C_i^2, ..., C_i^j\}$ is the set of worst-case execution times for the segments. When k is odd, C_i^k is the WCET of sequential segment s_k on an unit-speed processor. When k is even, C_i^k is the WCET of each thread in parallel segment s_k on an unit-speed processor. The segments with odd numbers are sequential and the task starts and ends with a sequential segment. An example of a fork/join task is shown in Figure 12(a).



(a) A fork-join task. (b) A parallel synchronous task.

Fig. 12: Parallel synchronous tasks.

The *parallel synchronous task model* extends the fork/join task by removing the constraint that odd segments must be sequential (Saifullah et al., 2011). Instead, the

number of threads in any segment is arbitrary. An example of this model is shown in Figure 12(b).

DAG Task Model

A *DAG parallel task* (Saifullah et al., 2011; Bonifaci, 2013; Jiang et al., 2017, 2016; Saifullah et al., 2012) models parallel workload structure with arbitrary directed acyclic graphs. A DAG parallel task τ_i is characterized with (C_i, D_i, T_i), where D_i and T_i are relative deadlines and minimum inter-release separations as before. $\mathsf{C}_i = (C_i^1, C_i^2, ..., C_i^j)$ records the WCETs of each thread of the task. The DAG parallel task has an internal structure modeled by a directed acyclic graph $G = (V, E)$, describing the precedence relationships among the threads. Each node $v \in V$ represents a thread with worst-case execution time $c(v)$ and each edge $e \in E$ represents the execution order of the two nodes it connects. A node is eligible for execution only when all its predecessors have completed execution. An example of the DAG parallel task model is shown in Figure 13. v_0 and v_1 can execute simultaneously since both of them have no procedessor nodes. v_2 can start execution only when all of its predecessor nodes (v_0 and v_1) complete.

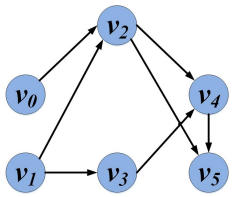


Fig. 13: A DAG task.

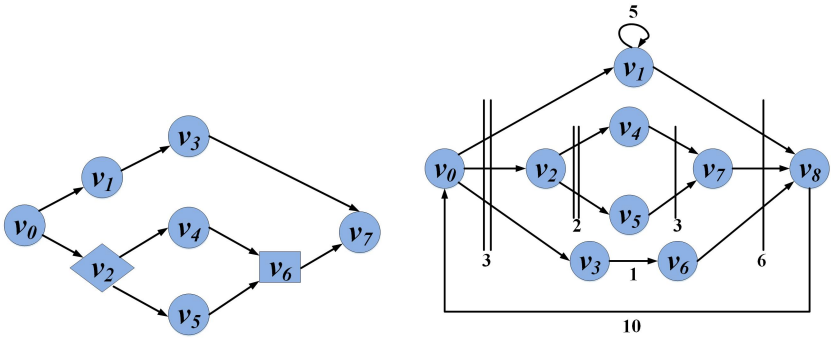
Both the DAG parallel task model and graph-based task models, e.g., the DRT model, introduced in last section use directed graphs to model the workload structure, but they are different in the following aspects. First, the multiple outgoing edges of a node have *forking* semantics in the DAG parallel task model but have *branchin*g semantics in DRT task model. In a DAG parallel task, all the successors of a node v will be released for execution after v is finished, while in a DRT task, only one of the successors will be eligible. Second, each edge in a DRT task is affiliated with a minimum inter-release separation, and a node is released only after such a time interval has passed since the release of its predecessor. In a DAG parallel task, all the successors become eligible for execution as soon as the predecessor finishes execution.

Parallel Conditional Task Models

Combining the graph-based task models in last section and the parallel task models in this section yields a more expressive representation which can model workload structures with both *branching* semantics and *forking* semantics, such as the workload generated by OpenMP programs.

One example of such combination is the *parallel conditional task model* (Baruah, 2015; Melani et al., 2015). Similar to a DAG parallel task, a *parallel conditional task* τ_i has a minimum inter-release separation T_i and a relative deadline D_i. The graph structure of a parallel conditional task has two types of nodes. The first type is *regular nodes*, which model the execution threads (similar to the nodes in DAG parallel tasks). The second type is *conditional nodes*, which represent the start/end of branching structures. A node v in either type is affiliated with a WCET value $c(v)$. The start node of a conditional branch is denoted by a diamond and the end node is denoted by a rectangle. A start node must be paired with an end node, and there is no edge pointing to nodes outside the branching structure defined by this pair of start/end nodes. An example of conditional DAG tasks is shown in Figure 14(a). The execution starts with v_0, and traverses along the edges. v_2 and v_6 are a pair of start/end nodes of branching structures. Only one path in $\{v_2, v_4, v_6\}$ and $\{v_2, v_5, v_6\}$ can be chosen to be traversed, since they are different branches in a branching structure and only one branch can be executed at one time.

The *fork-join real-time task model* (Stigge, Ekberg, & Yi, 2012; Sun et al. , 2016) is another way to combine the conditional branching and parallel workload structure. A *fork-join real-time task model* is represented by a hypergraph $G = (V,E)$. Each $v \in V$ denotes the release of a job labeled with the worst-case execution time $c(v)$ and relative deadline $d(v)$ as in DRT. There are two types of edges in the graph. One type is *sequential edges*, and the other type is *hyperedges*. A sequential edge is the same as an edge in the DRT model. A *hyperedge* is actually a set of edges starting with the same node or ending with the same node. A *fork hyperedge* $\{(u,v_1),(u,v_2),...,(u,v_i)\}$ is annotated with an intersecting double line in the graph (as shown in Figure 14(b)), and includes a set of edges which start with the same node, representing the start of multiple parallel tasks. Correspondingly, a *join hyperedge* $\{(u_1,v),(u_2,v),...,(u_i,v)\}$ is annotated with an intersecting single line and includes a set of edges which end with the same node, representing the end of parallel tasks and all joining back to the sequential part. A fork hyperedge must be paired with a join hyperedge, and the fork/join structure can be nested. All fork edges starting from the same node are labeled with same minimum inter-release separation and it is the same with join hyperedges. An example of the fork-join real-time tasks is shown in Figure 14(b). Assume that a job sequence is denoted by $J = (J_0, J_1, \cdots)$ with job parameters $J_k = (r_k, v_k)$ of release time r_k and job type v_k. A possible job sequence is $\{(0,v_0),(4,v_1),(3,v_2),(3,v_3),(5,v_4),(6,v_5),(9,v_7),(5,v_6)(15,v_8)\}$.



(a) A conditional DAG task. (b) A fork/join real-time task.

Fig. 14: Examples of parallel conditional tasks.

Data-Flow Task Model

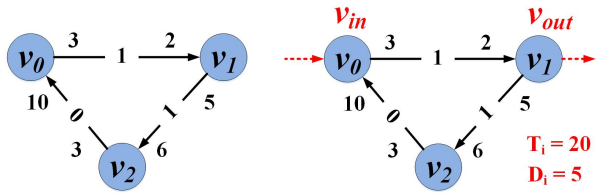
The data-flow graph models (Lee and Messerschmitt, 1987; Lee and Seshia, 2011; Singh and Baruah, 2017; Singh, Ekberg, & Baruah, 2017) not only represent parallel workload structures, but also can describe the amount of data consumed at the input and produced at the output of each computation unit. A widely used data-flow graph model is the *synchronous data-flow (SDF) graph model* (Lee and Messerschmitt, 1987). An SDF is represented by a directed graph $G = (V, E, PROD, CONS, DELAY)$. Each node $v \in V$ (called *actor*) represents a computation unit and each edge $e \in E$ (called *channel*) represents the direction of data flow from the output of one actor to the input of another actor. An actor consumes a certain amount data (tokens) from its input and produces a certain amount of tokens as output in one execution. An edge e starting with u and ending with v is labeled with $(PROD(e), CONS(e), DELAY(e))$ where $PROD(e)$ represents the number of tokens produced by one execution of u, $CONS(e)$ is the number of tokens consumed by one execution of v, and $DELAY(e)$ is the number of initial tokens in channel e before the system starts run.

An actor is eligible for execution as soon as there are enough amount of tokens on all its input channels. For node v, if every incoming edge e contains more than $CONS(e)$ tokens, v is *enabled*. After the node is executed, $CONS(e)$ tokens are removed from each of its input channel e and $PROD(e)$ tokens are added to each of its output channel e. An example of a synchronous data-flow task is shown in Figure 15(a).

The original SDF model does not involve any timing information. Some work (Bamakhrama and Stefanov, 2011, 2013; Spasic et al., 2015) extended the SDF model to model real-time systems by adding timing information. A *sporadic real-time SDF task model* was proposed (Singh and Baruah, 2017; Singh, Ekberg, & Baruah, 2017) by adding the following timing constraints to SDF. Each SDF task τ_i has a minimum inter-release separation T_i and a relative deadline D_i. Each actor

v is affiliated with a WCET $c(v)$. A fixed source actor v_{in} and sink node actor v_{out} are defined for each task. v_{in} is triggered when there are enough tokens on its input channel, including an external token which arrives with a minimum inter-arrival separation of T_i. The sink actor v_{out} must be finished no longer than D_i time units after the arrival of the external token at v_{in}.

An example of a sporadic real-time SDF task τ_i is shown in Figure 15(b), where v_0 is the source actor and v_1 is the sink actor. v_0 can start computation as long as the required amount of external token arrives and at least 10 tokens at its input on channel (v_2, v_0). The period is 20, enforcing the minimum inter-release separation between two consecutive v_{in}. If v_0 is triggered at t, the execution of v_1 in the same iteration has to be completed before $t+5$.



(a) A synchronous data-flow task.

(b) A sporadic real-time SDF task.

Fig. 15: An example of synchronous data-flow task model and its sporadic extension.

Real-Time Calculus

Real-Time Calculus (RTC) (Chakraborty, Knzli, & Thiele, 2003; Thiele, Chakraborty, Naedele , 2000) is a theoretical framework for performance analysis of networked embedded systems, which is rooted in the Network Calculus (Berten, Courbin , & Goossens, 2001). RTC uses *variability characterization curves* (called curves for short) (Wandeler, 2006) to model workload and resource. Comparing to the traditional real-time task model, the workload model is more general, and thus can model a much wider range of realistic systems. Two types of curves are defined to model workload in Real-Time Calculus. The first type is *cumulative function*, denoted by $R[s,\ t)$, which describes the accumulated incoming task requests during the time interval $[s,\ t)$. A cumulative function corresponds to a concrete event trace. The second type of curves is *arrival curves* defined as follows:

Definition 1 (Arrival Curve). Let $R[s,\ t)$ denote the total number of arrived events in time interval $[s,\ t)$, where s and t are two arbitrary non-negative real numbers. Then, the corresponding upper and lower arrival curves are denoted as α^u and α^l, respectively, and satisfy:

$$\forall s < t,\ \ \alpha^l(t-s) \leq R[s,t) \leq \alpha^u(t-s), \tag{1}$$

where $\alpha^u(0) = \alpha^l(0) = 0$.

Intuitively, arrival curves represent, for each Δ, the maximum and minimum number of events arrived in any time interval of length Δ. While cumulative function corresponds to one concrete event stream trace, arrival curves models the common worst-case/best-case timing behavior of a set of event stream traces whose cumulative functions are bounded in certain ranges. Figure 16(a) shows an event stream trace, its corresponding cumulative function and arrival curves. Events in a dashed box are released simultaneously. The arrival curves in RTC can be used to precisely or approximately represent the workload generated by many other real-time task models. For example, it was shown in Guan et al. (2015) how to derive the workload of a DRT task to utilize the analysis techniques in RTC.

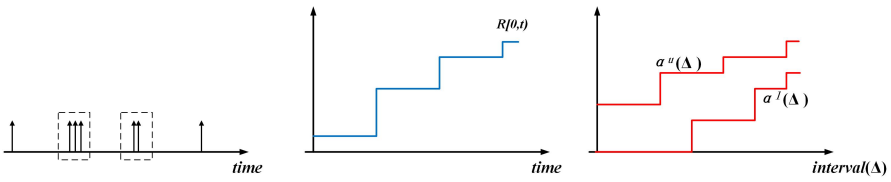


Fig. 16: An event sequence, its corresponding cumulative function and arrival curves.

Summary

This article gives a brief summary of some representative real-time task models, focusing on their expressiveness aspect. We first introduce the simple periodic/sporadic task model, and how it is extended to cover burst and jitter semantics. Second, we introduce the graph-based real-time task models describing more complex workload release patterns. What follows are parallel real-time task models describing parallel inter-task workload structures. Finally, arrival curves in Real-Time Calculus are introduced as a general workload representation.

References

J.A. Stankovic, Technical report, University of Massachusetts, 1992
C.L. Liu, J.W. Layland, Journal of the ACM (JACM) **20**, 46 (1973)
J.Y.-T Leung, M. Merrill, Information processing letters **11**, 115 (1980)
J.Y.-T Leung, J. Whitehead, Performance evaluation **2**, 237 (1982)
S.K. Baruah, A.K. Mok,L.E. Rosier, in *RTSS*, Lake Buena Vista, USA, Dec. 1990
J.P. Lehoczky, L. Sha, J.K. Strosnider, in *RTSS*, San Jose, USA, 1-3 Dec. 1987
A.K. Mok, Dissertation, Massachusetts Institute of Technology, 1983
B. Sprunt, L. Sha, J. Lehoczky, Real-time systems **1**, 27 (1989)
N. Audsley, A. Burns, M. Richardson, K. Tindell, A.J. Wellings, Softwawre Engineering Journal **8**, 284 (1993)
K.W. Tindell, A. Burns, A.J. Wellings, Real-time systems **6**,133 (1994)
K. Richter, R. Racu, R. Ernst, in *RTSS*, Cancun, Mexico, 3-5 Dec. 2003
J.C. Palencia, M.G. Harbour, in *RTSS*, Madrid, Spain, 2-4 Dec. 1998
J.C. Palencia, M.G. Harbour,in *ECRTS*, Porto, Portugal, 2-4 July 2003
K. Tindell, Technical report, University of York, 1994
A.K. Mok, D. Chen, IEEE transactions on software engineering **23**, 635 (1997)
S.K. Baruah, D. Chen, S. Gorinsky, A. Mok, Real-time systems **17**, 5 (1999)
H. Takada, K. Sakamura, in *RTCSA*, Taipei, Taiwan, 27-29 Oct. 1997
S.K. Baruah, in *RTCSA*, Macau, China, 23-25 Aug. 2010
N.T. Moyo, E. Nicollet, F. Lafaye, C. Moy, in *ECRTS*, Brussels, Belgium, 6-9 July 2010
S.K. Baruah, in *ECRTS*, Berlin, Germany, 17-19 June 1998
M. Anand, A. Easwaran, S. Fischmeister, I. Lee, in *ISORC*, Orlando, USA, 5-7 May 2008
S.K. Baruah, Real-time systems **24**, 93 (2003)
S. Chakraborty, T. Erlebach, L. Thiele, in *WADS*, Providence, USA, 15-17 Aug. 2001
S.K. Baruah, in *RTSS*, Madrid, Spain, 2-4 Dec. 1998
S.K. Baruah, in *RTSS*, San Diego, California, 30 Nov.-3 Dec. 2010
M. Stigge, Dissertation, Uppsala University, 2014
M. Stigge, P. Ekberg, N. Guan, W. Yi, in *RTAS*, Chicago, Illinois, 11-14 April 2011

M. Mohaqeqi, J. Abdullah, N. Guan, W. Yi, in *ECRTS*, Toulouse, France, 5-8 July 2016
M. Stigge, W. Yi, in *ECRTS*, Pisa, Italy, 11-13 July 2012
M. Stigge, W. Yi, in *RTSS*, Vancouver, Canada, 3-6 Dec. 2013
M. Stigge, W. Yi, in *ECRTS*, Madrid, Spain, 8-11 July 2014
M. Stigge, P. Ekberg, N. Guan, W. Yi, in *ECRTS*, Porto, Portugal, 6-9 July 2011
E. Fersman, P. Pettersson, W. Yi, in *TACAS*, Grenoble, France, 8-12 April 2002
J. Bengtsson, W. Yi, in *Lectures on concurrency and petri nets*, (Springer, New York), pp. 87-124
D.G. Feitelson, L. Rudolph, Journal of parallel and distributed computing **16**, 306 (1992)
V. Berten, P. Courbin, J. Goossens, in *RTNS*,Nantes, France, 29-30 Sep. 2011
Z. Dong, C. Liu, in *RTSS*, Paris, France, 5-8 Dec. 2017
K. Lakshmanan, S. Kato, R. Rajkumar, in *RTSS*, San Diego, USA, 30 Nov. -3 Dec. 2010
OpenMP, http://openmp.org.
J. Sun, N. Guan, Y. Wang, Q. He, W. Yi, in *RTSS*, Paris, France, 5-8 Dec. 2017
Y. Wang, N. Guan, J. Sun, M. Lv,Q. He, T. He,W. Yi, in *RTCSA*, Hsinchu, Taiwan, 16-18 Aug. 2017
A. Saifullah, K. Agrawal, C. Lu, C. Gill, in *RTSS*, Vienna, Austria, 29 Nov.-2 Dec. 2011
V. Bonifaci, A.M. Spaccamela, S. Stiller,A. Wiese, in *ECRTS*, Paris, France, 9-12 July 2013
X. Jiang, N. Guan, X. Long, W. Yi, in *RTSS*, Paris, France, 5-8 Dec. 2017
X. Jiang, X. Long, N. Guan, H. Wan, in *RTSS*, Porto, Portugal, 29 Nov.-2 Dec. 2016
A. Saifullah, D. Ferry, C. Lu, C. Gill. Technical report, Washington University, 2012
S.K. Baruah, in *EMSOFT*, Amsterdam, Netherlands, 4-9 Oct. 2015
A. Melani, M. Bertogna, V. Bonifaci, A.M. Spaccamela, in *ECRTS*, Lund, Sweden, 7-10 July 2015
M. Stigge, P. Ekberg, W. Yi, ACM SIGBED Review **10**, 20 (2012)
J. Sun, N. Guan, Q. Deng, P. Zeng, W. Yi, ACM transactions on embedded computing systems (TECS) **15**, 14:1 (2016)
E.A. Lee, D.G. Messerschmitt, Proceedings of the IEEE **75**, 1235 (1987)
E.A. Lee, S.A. Seshia, in *Introduction to embedded systems, a cyber-physical systems approach* (MIT Press, 2011)
A. Singh, S.K. Baruah, in *RTSS*, Paris, France, 5-8 Dec. 2017
A. Singh, P. Ekberg, S.K. Baruah, in *ECRTS*, Dubrovnik, Croatia, 27-30 June 2017
M. Bamakhrama, T. Stefanov, in *EMSOFT*, Taipei, Taiwan, 9-14 Oct. 2011
M. Bamakhrama, T. Stefanov, Design automation for embedded systems **17**, 221 (2013)
J. Spasic, D. Liu, E. Cannella, T.Stefanov, in *CODES+ISSS*, Amsterdam, Netherland, 4-9 Oct. 2015
S. Chakraborty, S. Knzli, L. Thiele, in *DATE*, Munich, Germany, 3-7 March 2003
L. Thiele, S. Chakraborty, M. Naedele, in *ISCAS*, Geneva, Switzerland, 28-31 May 2000

J.-Y. L. Boudec, P. Thiran, in *Network calculus: a theory of deterministic queuing systems for the internet* (Springer-Verlag Berlin Heidelberg, 2001)
E. Wandeler, Dissertation, ETH Zurich, 2006
N. Guan, Y. Tang, Y. Wang, W. Yi, in *DATE*, Grenoble, France, 9-13 March 2015

List of Figures