RD-Gen: Random DAG Generator Considering Multi-rate Applications for Reproducible Scheduling Evaluation

***Abstract*—Real-time systems have various requirements such as deadline and resource constraints. These systems are becoming larger and more complex, and study on performance analysis and efficient scheduling algorithms is becoming more important. A directed acyclic graph (DAG) model that can express task dependencies and parallelism is used for such studies. DAG researchers use a random DAG set to demonstrate the effectiveness and objectivity of their proposed method. However, there is no random DAG generation tool that can generate a DAG set that considers the latest multi-rate systems. Therefore, researchers must generate random DAGs on their own, which leads to secondary labor and reduced reliability and reproducibility. To solve this problem, We propose a multi-rate DAG generator (MRDAG-Gen) that considers the constraints and configurations of state-of-the-art multi-rate systems. MRDAG-Gen also provides batch generation of random DAG sets with different parameters. The case study shows that MRDAG-Gen covers various problem settings and DAG study requirements.**

***Index Terms*—DAG, Random generation tool, Multi-rate systems**

I. Introduction

Real-time systems such as self-driving systems must be successfully executed while meeting various requirements such as output within a pre-determined time (i.e., deadline), low power consumption, and resource constraints [1]–[3]. To meet these constraints, there has been much research on task allocation and scheduling [4]–[6], as well as on analyzing the end-to-end latency and the response time of a system [7]–[9]. Systems are becoming larger and more complex every year, and such studies use models that represent the complex dependencies and parallelism of tasks in the system, such as a directed acyclic graph (DAG).

DAGs are used in many allocation, scheduling, and latency analysis studies [10]–[12] because they express the flow of processing from system input to output and can represent various kinds of information such as dependencies between tasks, task execution time, and execution period. To evaluate the performance of proposed methods in these studies, it is important to compare them with existing methods using task sets. Then, in the evaluation of methods using DAGs, randomly generated DAGs are used to ensure objectivity and to demonstrate generality [2], [13], [14].

To ease such evaluations, random DAG generation tools such as the task graph for free (TGFF) [15] and GGen [16] have been proposed and utilized in the latest publications [17]–[20]. These tools allow the user to parametrically specify the shape of the DAG (e.g., number of tasks, in-degree and out-degree per task) and the properties assigned to tasks and edges (e.g., execution time, execution period, communication time). Furthermore, because these tools use a pseudo-random number generator, other researchers can easily reproduce the DAG set by specifying the same options. However, TGFF and GGen have been proposed in 1999 and 2010, respectively, and cannot meet the requirements for multi-rate DAGs considering the state-of-the-art real-time systems.

Since embedded systems in automobiles and avionics, as well as self-driving systems, contain multiple tasks that operate at different periods (e.g., sensors [21], localization [14] and angle synchronous [22]), research targeting multi-rate DAGs is becoming increasingly important [8], [23]. In studies of such multi-rate DAGs, not only the shape of the DAG but also the ratio of execution time to task execution period has a significant impact on the performance of the method (e.g., implicit deadline [24], [25] and task utilization [26], [27]). However, TGFF cannot generate multi-rate DAGs, and GGen can only randomly set the period and the execution time to tasks. Therefore, most authors who consider multi-rate DAGs have their implementation of a random DAG set [26]–[29]. It is laborious for researchers to prepare their own set of random DAGs, and further reduces the reliability and reproducibility of the evaluation results.

To solve these problems, this paper proposes a random DAG generation tool called a multi-rate DAG generator (MRDAG-Gen) that meets the requirements of state-of-the-art research. MRDAG-Gen extends existing DAG generation methods and provides a flexible evaluation platform. Since MRDAG-Gen uses a pseudo-random number generator, other researchers can reproduce the DAG sets used in the evaluation by specifying the same options. In addition, MRDAG-Gen supports researchers by providing batch generation of all random DAG sets at different parameters and the functionality to visualize scheduling results.

Contributions: Our primary contributions are summarized as follows.

- MRDAG-Gen provides a flexible DAG set by adding parameters to existing DAG generation methods and new chain-based generation methods.
- MRDAG-Gen automatically sets properties that meet implicit deadlines and utilization constraints.
- MRDAG-Gen reduces implementation effort through a batch generation of random DAG sets.

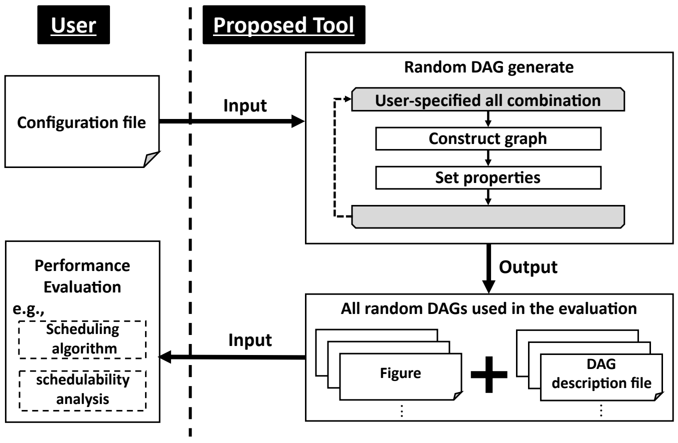


Fig. 1. System model.

TABLE I
DAG NOTATIONS

Symbols	Descriptions
G	DAG
V	Set of all nodes of G
$\|V\|$	Total number of nodes of G
E	Set of all edges of G
τ_i	i-th Node
C_i	WCET of τ_i
$e_{i,j}$	Edge between τ_i and τ_j
$comm_{i,j}$	Communication time of $e_{i,j}$
CCR	CCR of G
D	End-to-end deadline
τ_i^{tm}	i-th timer-driven node
ϕ_i	Offset of τ_i^{tm}
T_i	Execution period of τ_i^{tm}
d_i	Relative deadline of τ_i^{tm}
u_i	Utilization of τ_i^{tm}
U	Total Utilization of G
Γ_i	i-th chain
$\|\Gamma\|$	Total number of chains of G
C_{Γ_i}	WCET of Γ_i
u_{Γ_i}	Utilization of Γ_i
T_{Γ_i}	Period of head timer-driven node of Γ_i

The remainder of the paper is organized as follows. Section II describes a system model. Section III explains the design and implementation of MRDAG-Gen. Section IV presents case studies. Section V compares MRDAG-Gen with existing random DAG generation methods. Section VI discusses related work. Finally, Section VII presents the conclusions and future work.

II. SYSTEM MODEL

This section represents the system model, as shown in Fig. 1. Section II-A describes the basic single-rate DAG. Section II-B explains a multi-rate DAG. The DAG notations used in this paper are listed in Table I.

A. Single-rate DAG

Single-rate DAGs are DAGs with either a single entry node or all entering at the same time, used in real-time application [30], cyber-physical systems [2] and cloud computing [3], and so forth. Here, the entry node represents the input to the system (e.g., a sensor event or a command from the user), and the exit node represents the final output from the system.

A DAG consists of a node set and an edge set, denoted $G = (V, E)$. Nodes represent tasks in the system, and edges represent communication and dependencies between nodes or priority constraints. V is the set of all nodes, expressed as $V = \{\tau_1, ..., \tau_{|V|}\}$, where $|V|$ is the total number of nodes. Each node has a worst-case execution time (WCET), and the WCET of τ_i is denoted as C_i. E is the set of all edges, where each edge $e_{i,j} \in E$ represents communication between τ_i and τ_j and a priority constraint. When $e_{i,j}$ exists in the DAG, τ_j cannot be executed until τ_i has completed its execution and the output of τ_i has arrived. If the communication time is given as an assumption, the communication time at $e_{i,j}$ is denoted as $comm_{i,j}$. The ratio of the sum of the communication times of all edges to the sum of the execution times of all nodes is called the communication-to-computation ratio (CCR) and is defined by Eq 1.

$$CCR = \frac{\sum\limits_{e_{i,j} \in E} comm_{i,j}}{\sum\limits_{\tau_i \in V} C_i} \tag{1}$$

An end-to-end deadline D is set at the exit node when it is necessary to guarantee the safety of hard real-time systems [31] or the quality of service of cloud computing [32].

B. Multi-rate DAG

A multi-rate DAG is a DAG that contains multiple nodes that are triggered at different periods. Here, the definitions of nodes and edges in a multi-rate DAG are the same as those shown in Section II-A. Multi-rate DAGs can be broadly classified into two categories: (i) DAGs in which all nodes are timer-driven nodes, as in automotive systems [8], [14], and (ii) DAGs that combine a chain consisting of timer-driven nodes and a chain of linked event-driven nodes, as in self-driving systems [10], [33].

1) Multi-rate DAG consisting of only timer-driven nodes: Each timer-driven node in these multi-rate DAGs is denoted by τ_i^{tm}, and τ_i^{tm} is characterized by the tuple (ϕ_i, Ci, Ti, di). ϕ_i, T_i, and d_i represent the offset, execution period, and relative deadline, respectively. For DAGs that consider timer-driven nodes, every timer-driven node has a relative deadline of T_i time units indicating that every job of τ_i^{tm} has an absolute deadline at T_i time units after its release [25], [27]. Such a time constraint is called an implicit deadline, and MRDAG-Gen generates DAGs that consider implicit deadlines.

The utilization of τ_i^{tm} is denoted as u_i calculated by $u_i = C_i/T_i$. The total utilization U of a DAG consisting only of timer-driven nodes is defined by Eq. 2.

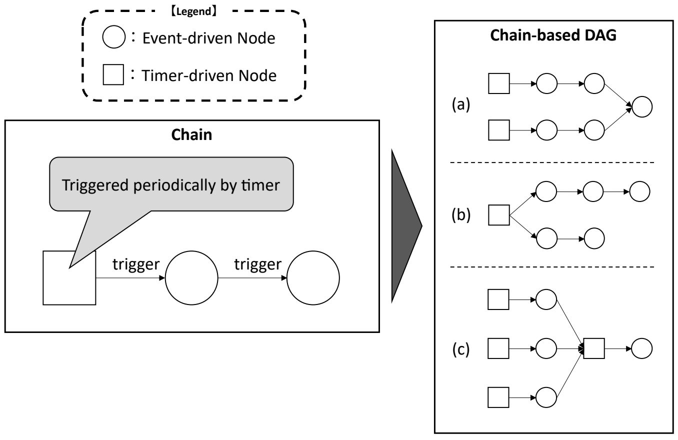


Fig. 2. Multi-rate DAGs consisting of chains

```
1   Seed: 0
2   Number of DAGs: 100
3
4   Graph structure:
5     Generation method: Fan−in/Fan−out
6     Number of nodes:
7       Combination: [10, 20]
8     In−degree:
9       Random: (start=1, stop=3, step=1)
10    Out−degree:
11      Random: (1, 3, 1)
12    Number of entry nodes:
13      Combination: [1, 3]
14    Number of exit nodes:
15      Fixed: 1
16
17  Properties:
18    Execution time:
19      Random: (1, 50, 1)
```

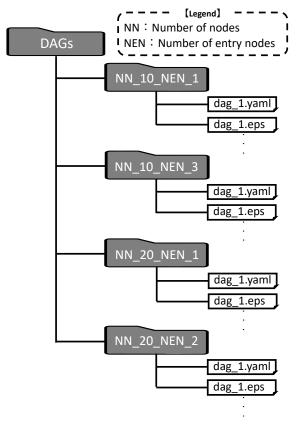


Fig. 3. Example of combination.

$$U = \sum_{\tau_i^{tm} \in V} u_i \tag{2}$$

2) Chain-based multi-rate DAG: In the latest multi-rate systems, DAGs consisting of the chain shown in Fig. 2 are considered. Each chain Γ_i is denoted as $\Gamma_i = \{\tau_i^{tm}, \tau_k, ..., \tau_{|\Gamma_i|}\}$, where $|\Gamma_i|$ is the number of nodes that compose Γ_i. The head τ_i^{tm} in the chain is triggered periodically, and subsequent event-driven nodes are triggered by their direct predecessors. This definition is also used in existing studies [33], [34]. The WCET of Γ_i is denoted by C_{Γ_i} and defined in Eq. 3.

$$C_{\Gamma_i} = \sum_{\tau_i \in \Gamma} C_i \tag{3}$$

Since the chain is executed dependent on the period of the head timer-driven node, the utilization of the chain u_{Γ_i} is calculated by Eq 4.

$$u_{\Gamma_i} = \frac{C_{\Gamma_i}}{T_{\Gamma_i}} \tag{4}$$

Here, T_{Γ_i} is the period of the head timer-driven node τ_i^{tm} of the chain. The total utilization of the chain-based multi-rate DAG is defined by Eq 5.

$$U = \sum_{\Gamma_i \in V} u_{\Gamma_i} \tag{5}$$

The chain-based multi-rate DAGs exist mainly in robot operating system (ROS)-based systems [10], [35]. In a typical ROS-based system, a self-driving system (e.g., Autoware [36]), different sensor data are processed and merged by multiple chains to output the final command. When modeling ROS-based systems as DAGs, it is necessary to consider DAGs where multiple chains merge at exit nodes ((a) in Fig. 2), where the chain branches ((b) in Fig. 2), and where multiple chains are vertically linked ((c) in Fig. 2). MRDAG-Gen can generate all these DAGs by using various parameters.

III. Design and implementation

This section describes the design and implementation of MRDAG-Gen. MRDAG-Gen iterates over the construction of the graph and the set of properties for all parameter combinations described by the user in the configuration file as shown in the upper right of Fig. 1. Section III-A shows the common functionalities of MRDAG-Gen, Section III-B describes the three graph generation methods of MRDAG-Gen, and Section III-C explains how to set properties for the graph.

A. Common functionality

MRDAG-Gen receives as input a YAML file written by the user, and with a single command, MRDAG-Gen generates all random DAGs with different parameter combinations. An example of an input parameter file in YAML format is shown on the left side of Fig. 3. Since MRDAG-Gen allows the user to specify seed values for pseudo-random number generation (line 1 in Fig 3), the same YAML file can be used to easily reproduce random DAG sets generated by other researchers. *Number of DAGs* (line 2 in Fig 3) is the number of DAGs randomly generated for each combination. The shape of the generated DAGs is determined by the parameters under *Graph structure*, and the properties set for nodes and edges are determined by the parameters under *Properties*.

For parameters that require numeric input, users can specify values in the following three ways.

1) *Fixed*: Only one value is specified, and this value is always used when the DAG is generated. This specification method can be utilized in DAGs where end-to-end deadlines are considered, and one exit node is always desired (line 15 in Fig. 3).
2) *Random*: When a DAG is generated, one is selected at random from the input range. The most basic specification method is used when there is no special intention and to have a variety of values, as shown in lines 9, 11 and 19 in Fig. 3.
3) *Combination*: MRDAG-Gen generates DAGs for all combinations of all lists of parameters for which *Combination* is specified. This specification method allows

all DAG sets used in random evaluations in DAG studies to be generated in single command execution. Therefore, MRDAG-Gen can significantly reduce the time and effort of researchers using DAGs. For example, when *Combination* of *Number of nodes* is *[10, 20]* and *Combination* of *Number of entry nodes* is *[1, 3],* MRDAG-Gen generates 100 DAGs for all combinations, as shown on the right side of Fig. 3.

When specifying ranges for *Random* and *Combination* in MRDAG-Gen, the following two intuitive descriptions are possible.

1) *List format*: The user describes the choices in a list (array) format (such as *[10, 20]* in the line 7 in Fig. 3)
2) *Tuple format*: The user describes *start* value, *stop* value, and *step* (such as *(start=1, stop=3, step=1)* in the line 9 in Fig. 3). The tuple format is internally expanded into a list format of values that are sequentially added to the *step* from the *start* value to the *stop* value (e.g., *(start=1, stop=3, step=1)* expands to a list of *[1, 2, 3]*). Here, *"start="*, *"stop="*, and *"step="* are optional (line 11 in Fig. 3).

MRDAG-Gen can output DAG description files in YAML, JSON, XML, and DOT formats, and DAG image files in EPS, PDF, SVG, and PNG formats.

B. Graph construction

MRDAG-Gen extends the *Fan-in/Fan-out* [15], *G(n, p)* [16] method widely used in the scheduling field for DAG researchers. In addition, MRDAG-Gen provides a *Chain-based* method for generating state-of-the-art chain-based multi-rate DAGs as shown in Fig 1. The parameters that can be specified in *Graph structure* of MRDAG-Gen are listed in Table III-A.

1) Fan-in/Fan-out method: Fan-in/Fan-out is the random DAG generation method proposed by Dick et al. and provided by TGFF [15]. *Fan-in/Fan-out* method can specify a range of in-degree (i.e., the number of edges to be input) and out-degree (i.e., the number of edges to be output) for one node, and a DAG is generated in which all nodes satisfy this condition. *Fan-in/Fan-out* method extends the graph by randomly repeating Fan-in and Fan-out phases. However, the original *Fan-in/Fan-out* method cannot completely satisfy the researcher's requirements.

Since scheduling and allocation methods using DAGs affect performance depending on the number of nodes in a DAG, evaluations are performed with various changes in the number of nodes [2], [6]. However, TGFF does not allow the user to completely control the number of nodes in a DAG to be generated. In addition, the DAG generated by *Fan-in/Fan-out* method has just one entry node, while in-vehicle and self-driving systems have multiple sensors [14], [21] and require a DAG with multiple entry nodes. Although the latest version of TGFF allows multiple entry nodes, it may generate DAGs that are not weakly connected. The number of exit nodes must also be able to be specified since many studies consider DAGs with a single exit node [25], [32].

Algorithm 1: *Fan-in/Fan-out* method in MRDAG-Gen

```
Input: $n \leftarrow$ Number of nodes
       $nen \leftarrow$ Number of entry nodes
       $nex \leftarrow$ Number of exit nodes
       $id \leftarrow$ In-degree
       $od \leftarrow$ Out-degree
Output: A DAG that satisfies user-specified parameters
1  Initialize $G \leftarrow (V, E)$, with $V \leftarrow \{\tau_1, ..., \tau_{nen}\}$ and $E \leftarrow \{\emptyset\}$
2  while $|V| \neq n - nex$ do
3      if Random(0, 1) = 1 then
           /* Fan-in phase */
4          $S \leftarrow$ Set of nodes in $V$ whose out-degree is less than or equal to $od$
5          $r \leftarrow$ Random(1, $id$)
6          $T \leftarrow$ Randomly choose $r$ nodes from $S$
7          $V \leftarrow V \cup \{\tau_{|V|+1}\}$
8          foreach $\tau_i \in T$ do
9              $E \leftarrow E \cup \{e_{i,|V|}\}$
10         end
11     end
12     else
           /* Fan-out phase */
13         $\tau_i \leftarrow$ The node with the largest difference between $od$ and its out-degree in $V$
14         $diff \leftarrow od -$ Out($\tau_i$)
15         $r \leftarrow$ Random(1, $diff$)
16         foreach $k \in |V|+1, ... |V|+m+1$ do
17             $V \leftarrow V \cup \{\tau_k\}$
18             $E \leftarrow E \cup \{e_{i,k}\}$
19         end
20     end
21     if $|V| > n - nex$ then
22         Initialize $G \leftarrow (V, E)$, with $V \leftarrow \{\tau_1, ..., \tau_{nen}\}$ and $E \leftarrow \{\emptyset\}$
23     end
24 end
25 $G \leftarrow$ add_exit_nodes($G$, $nex$)
26 $G \leftarrow$ weakly_connect($G$)
27 return $G$
```

Algorithm 2: add_exit_nodes(G, nex)

```
Input: $G$: A DAG, $nex$: Number of exit nodes
Output: DAG with exit nodes added
1  $X \leftarrow \{\tau_{|V|+1}, ..., \tau_{|V|+nex+1}\}$
2  $O \leftarrow$ Set of current exit nodes of $G$
3  while $\exists x \in X$, $In(\tau_x) = 0$, or $\exists o \in O$, $out(\tau_o) = 0$ do
4      $\tau_i \leftarrow$ The node with the smallest out-degree in $O$
5      $\tau_j \leftarrow$ The node with the smallest in-degree in $X$
6      $E \leftarrow E \cup \{e_{i,j}\}$
7  end
8  $V \leftarrow V \cup X$
9  return $G$
```

Algorithm 3: weakly_connect(G)

```
Input: $G$: A DAG
Output: A weakly connected DAG
1  while G is not weakly connected do
2      $M \leftarrow$ Weakly connected component with the maximum number of nodes in $G$
3      $W \leftarrow$ Randomly selected a weakly connected component other than $M$
4      $\tau_i \leftarrow$ Randomly choose one of the exit nodes in $W$
5      $\tau_j \leftarrow$ Randomly choose one node other than entry nodes in $M$
6      $E \leftarrow E \cup \{e_{i,j}\}$
7  end
8  return $G$
```

TABLE II
GRAPH STRUCTURE

Generation methods	Parameters		Descriptions
Fan-in/Fan-out G(n, p)	*Number of nodes*		Number of nodes in a single DAG
	Number of entry nodes		Number of entry nodes in a single DAG
	Number of exit nodes		Number of exit nodes in a single DAG
	Ensure weakly connected		When True is specified, the generated DAGs are always weakly connected
Fan-in/Fan-out	*In-degree*		Number of edges input to one node
	Out-degree		Number of edges output from one node
G(n, p)	*Probability of edge existence*		Probability of an edge present between any nodes
Chain-based	*Number of chains*		Number of chains in a single chain-based DAG
	Main sequence length		Length of the longest path in a single chain
	Number of sub sequences		Number of branches from the main sequence in a single chain
	Vertically link chains	*Number of entry nodes*	Number of entry nodes in a single DAG
		Main sequence tail	When True is specified, the tail of the main sequence is randomly connected to the head of another chain
		Sub sequence tail	When True is specified, the tail of the sub sequence is randomly connected to the head of another chain
	Merge chains	*Number of exit nodes*	Number of exit nodes in a single DAG
		Middle of chain	Merge from a tail node of a random chain to a node other than the head and tail of another chain
		Exit node	Merge a head node of a random chain with the exit node.

TABLE III
HELPER FUNCTIONS IN ALGORITHMS

Symbols	Descriptions
Random(a, b) $\mid a, b \in \mathbb{N}$	Function that returns a random natural number between a and b
In(τ_i)	Function that returns the in-degree of τ_i
Out(τ_i)	Function that returns the out-degree of τ_i
Sequence($\{\tau_i, ..., \tau_k\}$)	Function that returns a straight-line node sequence consisting of a set of input nodes. For example, when the input is $\{\tau_1, \tau_2, \tau_3\}$, return $\{\tau_1, e_{1,2}, \tau_2, e_{2,3}, \tau_3\}$.

To meet these requirements, MRDAG-Gen allows the specification of the number of nodes, entry nodes, and exit nodes in a single DAG and ensures weakly connected. The *Fan-in/Fan-out* method procedure provided by MRDAG-Gen is shown in Algorithm 1, and the helper functions used in algorithms are listed in Table III. First, the DAG is initialized with a specified number of entry nodes (line 1 in Algorithm 1). MRDAG-Gen iteratively expands the graph until the number of nodes fits within a user-specified value (line 2 in Algorithm 1). The graph is randomly extended in Fan-in phase or Fan-out phase (lines 3-20 in Algorithm 1). If the number of nodes exceeds the user-specified value, the DAG is initialized (lines 21-23 in Algorithm 1). After the loop ends, the *add_exit_nodes* function is called and nodes with no output edges are merged into a user-specified number of exit nodes with the minimum number of edges (Algorithm 2). Finally, the *weakly_connect* function is called to add edges until the DAG is weakly connected (Algorithm 3). With these extensions, MRDAG-Gen's *Fan-in/Fan-out* method allows the user to fully control the number of nodes, entry nodes, and exit nodes in one DAG and also guarantees weakly connected.

2) G(n, p) method: G(n, p) is a well-known graph generation method proposed by Paul Erdős, and Alfréd Rényi et al [16] that generates a graph by the number of nodes and the probability of edge existence between any nodes. Many of the random DAG sets used in the evaluation of the latest studies using DAGs are generated based on the *G(n, p)* method

Algorithm 4: *G(n, p)* method in MRDAG-Gen

```
   Input: $n \leftarrow$ Number of nodes
          $p \leftarrow$ probability of edge existence
          $nen \leftarrow$ Number of entry nodes
          $nex \leftarrow$ Number of exit nodes
   Output: A DAG that satisfies user-specified parameters
 1 $n \leftarrow n - nen - nex$
 2 Initialize $G \leftarrow (V, E)$, with $V \leftarrow \{\tau_1, ..., \tau_n\}$ and $E \leftarrow \{\emptyset\}$
 3 foreach $i \in 1, ..., |V|$ do
 4     foreach $j \in 1, ..., |V|$ do
 5         if $Random(0, 1) = 1$ and $i < j$ then
 6             $E \leftarrow E \cup \{e_{i,j}\}$
 7         end
 8     end
 9 end
10 $X \leftarrow \{\tau_{|V|+1}, ..., \tau_{|V|+nen+1}\}$
11 $O \leftarrow$ Set of current entry nodes of $G$
12 while $\exists x \in X$, $Out(\tau_x) = 0$, or $\exists o \in O$, $in(\tau_o) = 0$ do
13     $\tau_i \leftarrow$ The node with the smallest in-degree in $O$
14     $\tau_j \leftarrow$ The node with the smallest out-degree in $X$
15     $E \leftarrow E \cup \{e_{i,j}\}$
16 end
17 $V \leftarrow V \cup X$
18 $G \leftarrow$ add_exit_nodes($G$, $nex$)
19 $G \leftarrow$ weakly_connect($G$)
20 return $G$
```

[13], [28], [29], [37]. However, there are problems with the original *G(n, p)* method, such as the possibility of a cycle and the inability to guarantee a weakly connected.

Therefore, MRDAG-Gen extended the *G(n, p)* method for direct use in the evaluation of DAG studies. The procedure for the *G(n, p)* method in MRDAG-Gen is shown in Algorithm 4 First, nodes other than entry and exit nodes are added to the graph (lines 1-2 in Algorithm 4). Then, for any two nodes, the addition of an edge is tried with a probability of 1 in 2. Here, if the index of the destination node is smaller than the index of the source node, the edge addition is canceled (line 5 in Algorithm 4). This condition causes no cycles in the graph [28], [38]. After the loop ends, a user-specified number of entry nodes and a node with an in-degree of 0 are connected with the smallest edge. Finally, the *add_exit_nodes* and *weakly_connect* functions are called the same as in *Fan-*

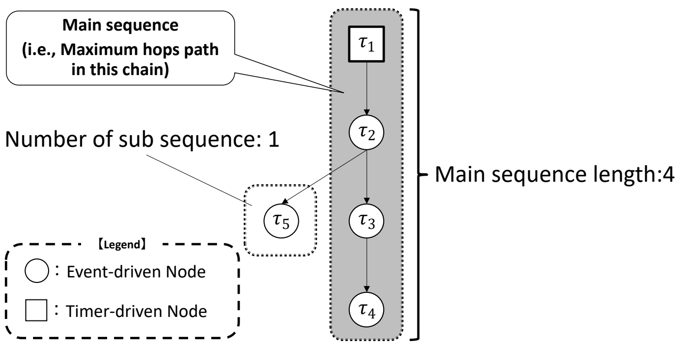


Fig. 4. Example of chain parameters

Algorithm 5: *Chain-based* method in MRDAG-Gen

```
   Input: $nc \leftarrow$ Number of chains
   $ml \leftarrow$ Main sequence length
   $ns \leftarrow$ Number of sub sequences
   $nen \leftarrow$ Number of entry nodes
   $nex \leftarrow$ Number of exit nodes
   Output: A DAG that satisfies user-specified parameters
1  Initialize $G \leftarrow (V, E)$, with $V \leftarrow \{\emptyset\}$ and $E \leftarrow \{\emptyset\}$
2  foreach $i \in 1, ..., nc$ do
       /* Construct each chain */
3      $main \leftarrow$ Sequence($\{\tau_{|V|+1}, ..., \tau_{|V|+ml+1}\}$)
4      $G \leftarrow main$
5      foreach $j \in 1, ..., ns$ do
6          $r \leftarrow$ Random($|V|+1, |V|+ml$)
7          $sl \leftarrow$ Random($1, ml - r$)
8          $sub \leftarrow$ Sequence($\{\tau_{|V|+1}, ..., \tau_{|V|+sl+1}\}$)
9          $E \leftarrow E \cup \{e_{r,|V|+1}\}$
10         $G \leftarrow sub$
11     end
12 end
   /* Vertically link chains */
13 while Number of entry nodes of $G \neq nen$ do
14     $\tau_i \leftarrow$ Randomly choose one of the exit nodes in $V$
15     $\tau_j \leftarrow$ Randomly choose one node from the head of the chains
16     $E \leftarrow E \cup \{e_{i,j}\}$
17 end
   /* Merge chains */
18 while Number of exit nodes of $G \neq nex$ do
19     $\tau_i \leftarrow$ Randomly choose one of the exit nodes in $V$
20     $\tau_j \leftarrow$ Randomly choose one node from other than head of the
         chains
21     $E \leftarrow E \cup \{e_{i,j}\}$
22 end
23 return $G$
```

in/Fan-out method.

3) Chain-based method: The *Chain-based* method is a new random DAG generation method to meet the requirements of modern chain-based multi-rate DAGs. In the *Chain-based* method, multiple chains are combined to construct a single DAG. The parameters that determine the shape of a single chain are explained here using Fig. 4. The user can determine the shape of one chain by specifying *main sequence length* and *Number of subsequences*. The main sequence is the path that has the largest number of hops from the head timer-driven node to the tail event-driven node in a single chain ($\{\tau_1, \tau_2, \tau_3, \tau_4\}$ in Fig. 4), and subsequences are the straight line of nodes branching off from the main sequence ($\{\tau_5\}$ in Fig. 4).

Algorithm 6: Set utilization

```
   Input: $G \leftarrow$ DAG
   $pt \leftarrow$ Peridoic type
   $U \leftarrow$ Total utilization
   $p \leftarrow$ Period
   Output: A DAG that satisfies user-specified parameters
1  if $pt =$ "All" then
2      $u[1, ..., |V|] \leftarrow$ UUniFast($|V|, U$) [39]
3      foreach $i \in 1, ..., |V|$ do
4          $T_i \leftarrow p$
5          $C_i \leftarrow u[i] \times T_i$
6      end
7  end
8  if $pt =$ "Chain" then
9      $u[1, ..., |\Gamma|] \leftarrow$ UUniFast($|\Gamma|, U$)
10     foreach $i \in 1, ..., |\Gamma|$ do
11         $T_{\Gamma_i} \leftarrow p$
12         $C_{\Gamma_i} \leftarrow u[i] \times T_{\Gamma_i}$
13         $g \leftarrow$ Number of nodes in $\Gamma_i$
14         $c[i, ..., i+g] \leftarrow$ Grouping randomly into $g$ with total equal to
             $C_{\Gamma_i}$
15         foreach $j \in i, ..., i+g$ do
16             $C_j \leftarrow c[j]$
17         end
18     end
19 end
20 return $G$
```

In the *Chain-based* method, two different ways of connecting chains can be specified. The first is to link chains vertically, i.e., the event-driven node at the end of each chain is randomly connected to the timer-driven node at the head of the other chain (such as the DAG on the lower right of Fig. 2). The second method is to integrate multiple chains in one specific node. The *Chain-based* method allows nodes other than the head node of the chain to be specified as integration nodes, and the event-driven node at the tail of each chain is merged into a randomly selected integration node (such as the DAG on the upper right of Fig. 2).

The procedure of the *Chain-based* method is shown in Algorithm 5. First, the chains are constructed for *Number of chains* specified by the user (lines 2-12 in Algorithm 5). In the construction of each chain, the main sequence is first generated (lines 3-4 in Algorithm 5). Subsequences branch from random nodes other than the tail of the main sequence and are coordinated not to exceed the length of the main sequence (lines 6-10 in Algorithm 5). After all chains are generated, the chains are linked vertically until the number of entry nodes in the DAG is equal to the specified number (lines 13-17 in Algorithm 5). Finally, the chain is randomly merged until the number of exit nodes in the DAG becomes the specified number (lines 18-22 in Algorithm 5). Thus, the *Chain-based* method can flexibly generate DAGs consisting of multiple chains.

C. Property setting

MRDAG-Gen automatically sets the typical properties that characterize the nodes and edges of a DAG to meet user requirements. All properties that can be set automatically in MRDAG-Gen are listed in Table III-B3. MRDAG-Gen can generate (i) single-rate, (ii) multi-rate DAGs consisting of only

TABLE IV
PROPERTIES

Parameters			Descriptions
Execution time			Execution time of nodes
Communication time			Communication time of edges
CCR			CCR of a single DAG
End-to-end deadline	*Ratio of deadline to critical path*		Ratio of the end-to-end deadline to the critical path of DAG
Multi-rate	*Periodic type*		Specify a group of nodes to be timer driven
	Period		Period of timer-driven nodes
	Entry node period		Period of entry timer-driven nodes
	Exit node period		Period of exit timer-driven nodes
	Offset		Offset of timer-driven nodes
	Total utilization		Total utilization of a single DAG
	Maximum utilization		Maximum utilization of one timer-driven node
Additional properties	*Node properties*		User-defined numeric parameters with any name for nodes
	Edge properties		User-defined numeric parameters with any name for edges
Output formats	*DAG*	*YAML*	Outputs DAG description files in the format specified by True
		JSON	
		XML	
		DOT	
	Figure	*Draw legend*	When True is specified, a legend is drawn on the output DAG figure
		PNG	Outputs DAG figures in the format specified by True
		SVG	
		EPS	
		PDF	

timer-driven nodes, and (iii) chain-based multi-rate DAGs, depending on the *Periodic type* parameter. If the *Periodic type* is not specified, DAGs of (i) are generated; if the *Periodic type* is *"All"*, DAGs of (ii) are generated; if the *Periodic type* is *"chain"*, DAGs of (iii) are generated.

In MRDAG-Gen, all properties described in Section II can be specified using *"Fixed"*, *"Random"* or *"Combination"*. Although existing random DAG generation tools such as TGFF and GGen provide the functionality to randomly assign properties, they do not allow the user to control the values calculated by multiple properties such as CCR and total utilization. Since CCR and total utilization have a significant impact on the performance of scheduling algorithms and analysis methods, evaluations are performed by varying these values [2], [13], [38], [40]. Therefore, MRDAG-Gen also supports the specification of such complex property values.

An example of a property setting based on total utilization in MRDAG-Gen is shown in Algorithm 6. MRDAG-Gen uses the UUniFast method [39] to uniformly assign utilization to each node (lines 2 and 9 in Algorithm 6). If the *Periodic type* is *"All"*, the utilization for each node is determined based on the *Total utilization* specified by the user, and the period and execution time are set to satisfy this utilization (lines 1-7 in Algorithm 6). Here, when *Maximum utilization* is specified, MRDAG-Gen allocates utilization to each node not to exceed *Maximum utilization*. If the *Periodic type* is *"Chain"*, the utilization of each chain is determined based on the *Total utilization*, and the period of each chain and the sum of the execution time are calculated accordingly (lines 8-12 in Algorithm 6). The total execution time is randomly divided by the number of nodes in the chain to set the execution time for each node (lines 13-17 in Algorithm 6). In this way, MRDAG-Gen randomly and automatically sets properties according to complex parameters specified by the user.

MRDAG-Gen provides other parameters that allow for setting the ratio of end-to-end deadlines to critical path length and the period of entry and exit nodes. In addition, users can define their unique parameters for simple properties that uniformly assign numerical values to nodes or edges.

IV. CASE STUDY

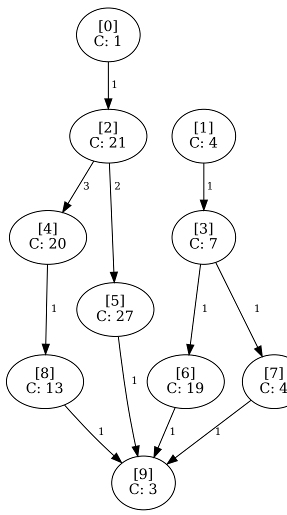
This section illustrates that MRDAG-Gen can generate the random DAG sets used in the evaluation of existing studies based on DAGs. Case studies of a single-rate DAG, a multi-rate DAG consisting of only timer-driven nodes, and a chain-based multi-rate DAG are shown, respectively.

A. Case study 1

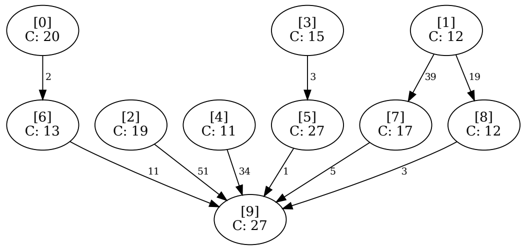
Since CCR changes the nature of DAGs and affects the performance of scheduling algorithms, existing studies of single-rate DAGs have used random DAG sets with different CCR values in their evaluations. An example of the generation in MRDAG-Gen of a random DAG set with varying CCR, number of nodes, execution time, and communication time using the *Fan-in/Fan-out* method as used in existing studies [41]–[43] is shown in Fig. 5. Since the *Combination* is specified for the *Number of nodes* (lines 6-7 in Fig. 5) and *CCR* (lines 21-22 in Fig. 5), MRDAG-Gen generates 100 random DAGs each for all combinations of these parameters (i.e., $\{10, 20, 30, ..., 1000\} \times \{0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0\}$). For each DAG, after graph construction, execution time is randomly assigned to each node in the range of 1 to 30 (lines 19-20 in Fig. 5). Then, the total communication time is calculated from the CCR and the sum of execution time based on Eq. 1, and the total communication time is randomly distributed to each edge. Thus, the user can generate all DAG sets used in the evaluation with a single command, without adjusting the number of nodes or CCR.

```
Seed: 0
Number of DAGs: 100

Graph structure:
  Generation method: Fan−in/Fan−out
  Number of nodes:
    Combination: (10, 1000, 10)
  In−degree:
    Random: [1, 2, 3]
  Out−degree:
    Random: [1, 2, 3]
  Number of entry nodes:
    Random: [1, 2, 3, 4, 5]
  Number of exit nodes:
    Fixed: 1
  Ensure weakly connected: True

Properties:
  Execution time:
    Random: (1, 30, 1)
  CCR:
    Combination: [0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0]
```

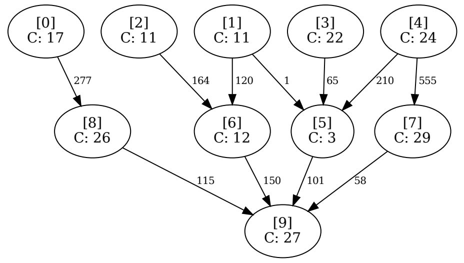
Input YAML file



Number of nodes: 10, CCR: 0.1



Number of nodes: 10, CCR: 1.0



Number of nodes: 10, CCR: 10.0

Fig. 5. Example of the generation a random DAG set with varying CCR, number of nodes, execution time, and communication time using the *Fan-in/Fan-out* method.

B. Case study 2

Random DAG sets with different total utilization are used in the evaluation of most studies considering multi-rate DAGs. An example of generating a multi-rate DAG with a random period and execution time for each node based on the total utilization of the DAG, as used in existing studies [13], [23], [24], is shown in Fig 6. Here, timer-driven nodes are drawn as squares. Since *Combination* is specified for *Total utilization* (lines 21-22 in Fig 6), MRDAG-Gen generates 100 DAGs each with total utilization from 5% to 95% in 5% increments. For each DAG, the utilization of each node is determined by the UUniFast method to satisfy the total utilization, and the execution time is calculated from the utilization and a randomly selected period from a uniform distribution ranging from 1 to 100.

C. Case study 3

In the evaluation of chain-based multi-rate DAGs, a random DAG set is used with varying total utilization for the entire system and each chain. An example of generating a random DAG set, as used in existing studies [10], [33], in which the utilization of each chain is determined from the total utilization of the entire system, and the period and execution time of each node is randomly assigned to satisfy the utilization of the chain is shown in Fig. 7. MRDAG-Gen randomly sets the utilization of each chain using the UUniFast method to achieve a determined total utilization rate. Here, since *Maximum utilization* is specified as 1.0 (line 24 in Fig. 7), MRDAG-Gen never generates a chain with a utilization greater than 1.0. The period of each chain is randomly set in the range of 50 to 1000, and the execution time of each node is calculated based on the utilization and period (lines 10-18 in Algorithm 6).

TABLE V
PROCESSES THAT MUST BE IMPLEMENTED BY THE USER AND LINES OF CODE

Symbols	Processes	Lines of code
(a)	Loop with different parameter values	3
(b)	Separate DAGs generated for each parameter into directories	5
(c)	Load .tgff file as DAG	40
(d)	Construct a DAG using G(n, p) method	10
(e)	Construct a DAG using chain-based method	80
(f)	Merge the exit nodes to a specific number	10
(g)	Add edges to be weakly connected	15
(h)	Set the properties randomly within a specific range	10
(i)	Set the execution time for each node and the communication time for each edge based on the CCR	25
(j)	Set the utilization for a multi-rate DAG consisting of only timer-driven nodes (lines 1-7 in Algorithm 6)	30
(k)	Set the utilization for a chain-based multi-rate DAG (lines 8-19 in Algorithm 6)	50

V. Evaluation

This section shows that compared to existing random DAG generation tools, MRDAG-Gen can generate a random DAG set with fewer lines of code and no unique user implementation. Table V shows the processes that must be implemented by the user and the number of lines of code used in the tool comparison. Here, the process in Table V is written in Python and shell scripts, and the Python NetworkX library is used for processes related to DAGs.

The amount of description required to generate a random DAG set for the case studies in Section IV-A is shown in Table VI. In this case, while TGFF and GGen require the user to implement more than 50 lines, MRDAG-Gen can generate all random DAG sets using only the tool's functionality. Since TGFF and GGen do not have the functionality to generate DAGs with different parameter settings at once, the user must execute the generation command many times while changing

```
Seed: 0
Number of DAGs: 100

Graph structure:
  Generation method: G(n, p)
  Number of nodes:
    Random: (10, 100, 10)
  Probability of edge:
    Random: (0.1, 0.9, 0.1)
  Number of entry nodes:
    Random: [1, 2, 3, 4, 5]
  Number of exit nodes:
    Random: [1, 2, 3, 4, 5]
  Ensure weakly connected: True

Properties:
  Multi−rate:
    Periodic type: ”All”
    Period:
      Random: (1, 100, 1)
    Total utilization:
      Combination: (0.05, 0.95, 0.05)
```

Input YAML file

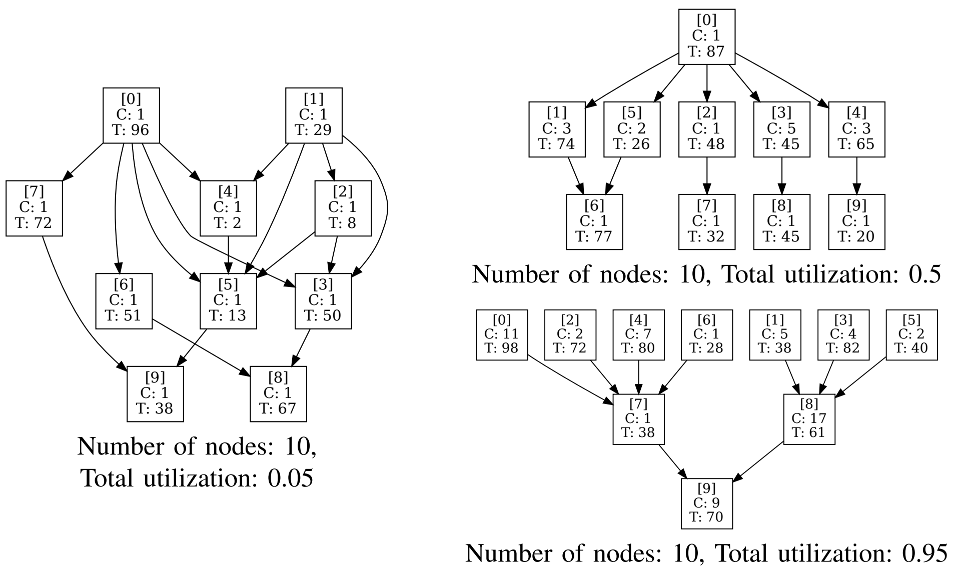

Number of nodes: 10,
Total utilization: 0.05
Number of nodes: 10, Total utilization: 0.5
Number of nodes: 10, Total utilization: 0.95

Fig. 6. Example of generating a multi-rate DAG consisting of only timer-driven nodes of various total utilization.

```
Seed: 0
Number of DAGs: 100

Graph structure:
  Generation method: Chain−based
  Number of chains:
    Random: [2, 3, 4, 5, 6, 7, 8, 9, 10]
  Main sequence length:
    Random: (2, 7, 1)
  Merge chains:
    Number of exit nodes:
      Random: [2, 3, 4, 5]
    Middle of chain: False
    Exit node: True

Properties:
  Multi−rate:
    Periodic type: ”Chain”
    Period:
      Random: (50, 1000, 1)
    Total utilization:
      Combination: (0.5, 4.0, 0.5)
    Maximum utilization:
      Fixed: 1.0
```

Input YAML file

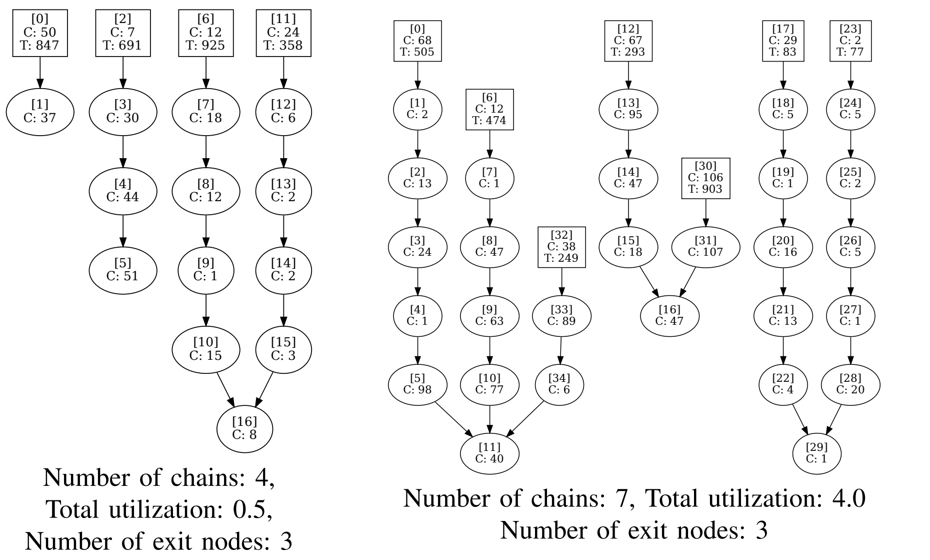


Fig. 7. Example of generating a multi-rate DAG consisting of only timer-driven nodes of various total utilization.

TABLE VI
EFFORT REQUIRED TO GENERATE RANDOM DAG SETS FOR *Case Study 1* [41]–[43]

	Lines of description for tool options	Lines of code
TGFF [15]	6	$(a) \times 2 + (b) + (c) + (f) + (g) + (i)$ $= 101$
GGen [16]	3	$(a) \times 2 + (b) + (g) + (i)$ $= 51$
MRDAG-Gen	20	0

the parameters, using shell scripts or other means. In contrast, MRDAG-Gen generates DAGs with all parameter combinations by specifying *Combination* as a parameter and automatically divides them into directories. Although TGFF uses the *Fan-in/Fan-out* method to construct DAGs, the .tgff files output by TGFF are in a proprietary format and require users to implement a reading process. Furthermore, since TGFF and GGen do not guarantee that the graph is weakly connected, users may unintentionally use a non weakly connected DAG for evaluation. TGFF and GGen cannot specify parameters such as CCR that are calculated from multiple properties. Therefore, the CCR must be adjusted for the DAG set after it is generated. In MRDAG-Gen, users can automate all of the above processes by inputting the YAML file as shown on the left side of Fig. 5.

The effort required to generate a random DAG set for the case studies in Section IV-B is shown in Table VII. TGFF does not support the *G(n, p)* method and multi-rate DAG. GGen allows properties to be randomly set to nodes or edges within a user-specified range after graph construction.

TABLE VII
EFFORT REQUIRED TO GENERATE RANDOM DAG SETS FOR *Case Study 2* [13], [23], [24]

	Lines of description for tool options	Lines of code
TGFF [15]	-	$(a) + (b) + (d) + (g) + (j)$ $= 63$
GGen [16]	2	$(a) + (b) + (g) + (j)$ $= 53$
MRDAG-Gen	20	0

TABLE VIII
EFFORT REQUIRED TO GENERATE RANDOM DAG SETS FOR *Case Study 3* [10], [33]

	Lines of description for tool options	Lines of code
TGFF [15]	-	$(a) + (b) + (d) + (k)$ $= 138$
GGen [16]	-	$(a) + (b) + (d) + (k)$ $= 138$
MRDAG-Gen	22	0

TABLE IX
MRDAG-GEN VS EXISTING METHODS

	RMD	RPU	RDT	RTE	OCG
DAGEN [44]			✓		
MRTG [45]			✓		
TGFF [15]			✓		
GGen [16]	✓		✓		
Kordon et al. [8]	✓				
Yang et al. [27]	✓	✓			
Gunzel et al. [23]	✓	✓			
Ueter et al. [24]	✓	✓			
Dong et al. [29]	✓	✓			
He et al. [13]	✓	✓			
Voronov et al. [28]	✓	✓			
Verucchi et al. [14]	✓	✓			
Klaus et al. [12]	✓	✓			
Tang et al. [33]	✓	✓		✓	
Choi et al. [10]	✓	✓		✓	
MRDAG-Gen	✓	✓	✓	✓	✓

RMD: Random generation of multi-rate DAGs
RPU: Random property settings based on total utilization
RDT: Random DAG generation tool
RTE: Random generation of chain-based multi-rate DAG
OCG: One-command batch generation of random DAGs

However, GGen does not allow users to specify the total utilization used in the evaluation of most studies that consider multi-rate DAGs. Therefore, the user has to go through the trouble of implementing such as lines 1-7 in Algorithm 6. In contrast, MRDAG-Gen uses the UUniFast method to randomly and automatically set the period and execution time for each node to meet the specified total utilization.

The amount of description to generate a chain-based random DAG set, as in the case study in Section IV-C, is shown in Table VIII. Chain-based multi-rate DAGs cannot be generated by TGFF and GGen because they are considered in state-of-the-art self-driving systems research. MRDAG-Gen can generate a batch of random chain-based DAG sets with different total utilization without any user implementation by inputting a YAML file as shown in Fig. 7.

The evaluation results demonstrated that MRDAG-Gen can generate the random DAG sets used in the evaluation with only an intuitive YAML file description, regardless of the single rate multi-rate. Therefore, MRDAG-Gen is a flexible evaluation platform that can be adapted to various requirements and provides reliability and reproducibility for the latest DAG studies.

VI. RELATED WORK

This section describes existing random DAG generation tools and existing studies using random DAGs and compares them to MRDAG-Gen. Table IX compares MRDAG-Gen and existing methods.

A. Random DAG generation with unique implementation

The random DAG generation tool provides reliability and reproducibility for the evaluation of scheduling and latency analysis studies. TGFF [15] is the first tool proposed for this purpose and has been used to evaluate the most recent studies [46]–[49]. TGFF can determine the shape of the DAG mainly by specifying the maximum and/or minimum input degree and maximum (first) output degree for one node (*Fan-in/Fan-out* method). TGFF can quickly generate many DAGs, and the task set can be easily reproduced by other researchers by inputting the same parameters. However, TGFF is a tool released in 1998 and has many problems, such as its output format (.tgff), which is difficult to handle and cannot generate the multi-rate DAGs.

GGen [16] is a unified implementation of the classical task graph generation methods used in the scheduling domain. GGen allows the user to add properties such as execution period and communication time to nodes and edges after generating DAGs using a user-specified generation method. However, GGen does not allow the specification of constraints between different properties, such as implicit deadlines (i.e., the execution time of a node must not exceed its period). Therefore, users with such requirements must adjust the values themselves.

Other random DAG generation tools such as DAGGEN [44] and MRTG [45] have been proposed. DAGGEN generates random workflow applications by specifying node load balancing, edge connection probability, and workflow shape. MRTG is a random DAG generation tool with a module-based implementation for user extensibility. However, these tools are not capable of generating multi-rate DAGs. In contrast, MRDAG-Gen can flexibly generate multi-rate DAGs of various types. In addition, MRDAG-Gen can automatically set properties calculated by multiple values such as CCR and total utilization.

B. Unique implementation of random DAG generation

This section presents existing studies in which the authors generate random DAG sets for evaluation in their own implemented algorithms and settings. Since there are no random DAG generation tools capable of generating multi-rate DAGs as described in Section VI-A, most studies that consider multi-rate DAGs have unique implementations.

In real-time systems such as in-vehicle systems and self-driving systems, multi-rate DAGs consisting of only timer-driven nodes are considered. Many real-time researchers use the *G(n, p)* method to construct DAGs and generate random task sets with the different number of nodes, different total utilization, and different periods. [13], [28], [29]. While proprietary algorithms are used to construct the graph in some cases, the approach is similar in that total utilization is determined using the UUniFast method and WCET of nodes are assigned based on utilization and periods [23], [24], [27].

In a multi-rate DAG considering automotive systems, a period is randomly assigned to each node based on the period observed in the automotive application. Verucchi et al. [14] randomly extended the automotive benchmark proposed by BOSCH in the 2015 WATERS Challenge [50] to analyze the performance of the proposed method. Verucchi et al. randomly set the task period from $[1, 5, 10, 20, 50, 100, 200, 1000]$ ms, as found in automotive applications, DAG utilization. Klaus et al. [12] set the utilization, period, and the number of nodes for each node of the DAG. Verucchi et al. and Klaus et al. create random task sets based on node chains consisting of only timer-driven nodes. MRDAG-Gen can also cover such chains consisting of only timer-driven nodes by using *Chain-based* methods and specifying the *Period type* as *"All"*. Kordon et al. [8] randomly set the period, the number of edges per task, release time, and the number of entry nodes for DAGs generated by the Python networkX library. MRDAG-Gen can automatically set all combinations of total utilization, periods, and the number of nodes as described above.

Studies of chain-based multi-rate DAGs, such as the latest ROS-based systems, have also been evaluated with random task sets. Tang et al. [33] allocate utilization to each chain based on the total utilization of the entire system and the number of chains, and then assign the utilization to the execution units in the chain. Choi et al. [10] similarly assign a utilization to each chain using the UUniFast method from the total system-wide utilization. Multi-rate DAGs based on such chains can be generated flexibly with the *Chain-based* method in MRDAG-Gen. In addition, since MRDAG-Gen also provides the functionality to automatically set the utilization corresponding to the chain, researchers do not need to implement it on their own.

VII. Conclusion

In this paper, we proposed a multi-rate DAG generator MRDAG-Gen that covers single-rate DAGs and state-of-the-art multi-rate DAGs. MRDAG-Gen extended the existing random graph generation method for DAGs and also provided a new chain-based method. MRDAG-Gen automatically set complex properties such as CCR and utilization. Moreover, MRDAG-Gen supported researchers by providing functions such as batch generation of all DAG sets for different parameters. Case studies showed that MRDAG-Gen meets the requirements of DAG studies in a variety of problem settings. Therefore, MRDAG-Gen provided reliability and easy reproducibility for DAG studies. In future work, We plan to extend MRDAG-Gen to cover more graph generation methods and complex properties.

References

[1] Ryotaro Koike and Takuya Azumi. Federated scheduling in clustered many-core processors. In *Proc. of DS-RT*, 2021.

[2] Debabrata Senapati, Arnab Sarkar, and Chandan Karfa. HMDS: A makespan minimizing DAG scheduler for heterogeneous distributed systems. *TECS*, 2021.

[3] Avinash Kaur, Parminder Singh, Ranbir Singh Batth, and Chee Peng Lim. Deep-Q-learning-based heterogeneous earliest finish time scheduling algorithm for scientific workflows in cloud. *J. Software. Pract. Exper.*, 2020.

[4] Shingo Igarashi, Takuro Fukunaga, and Takuya Azumi. Accurate contention-aware scheduling method on clustered many-core platform. *J. IPSJ*, 2021.

[5] Ali Asghari, Mohammad Karim Sohrabi, and Farzin Yaghmaee. Online scheduling of dependent tasks of cloud's workflows to enhance resource utilization and reduce the makespan using multiple reinforcement learning-based agents. *J. Soft. Comput.*, 2020.

[6] Zhao Tong, Xiaomei Deng, Hongjian Chen, Jing Mei, and Hong Liu. QL-HEFT: a novel machine learning scheduling scheme base on cloud computing environment. *J. Neural. Comput. Appl.*, 2020.

[7] Yuqing Yang and Takuya Azumi. Exploring real-time executor on ROS 2. In *Proc. of ICESS*, 2020.

[8] Alix Kordon and Ning Tang. Evaluation of the age latency of a real-time communicating system using the LET paradigm. In *Proc. of ECRTS*, 2020.

[9] Peng Chen, Hui Chen, Jun Zhou, Di Liu, Shiqing Li, Weichen Liu, Wanli Chang, and Nan Guan. Partial order based non-preemptive communication scheduling towards real-time networks-on-chip. In *Proc. of SAC*, 2021.

[10] Hyunjong Choi, Yecheng Xiang, and Hyoseung Kim. PiCAS: New design of priority-driven chain-aware scheduling for ROS2. In *Proc. of RTAS*, 2021.

[11] Viet Anh Nguyen, Damien Hardy, and Isabelle Puaut. Cache-conscious off-line real-time scheduling for multi-core platforms: algorithms and implementation. *J. Real-Time Syst.*, 2019.

[12] Tobias Klaus, Matthias Becker, Wolfgang Schröder-Preikschat, and Peter Ulbrich. Constrained data-age with job-level dependencies: How to reconcile tight bounds and overheads. In *Proc. of RTAS*, 2021.

[13] Qingqiang He, Mingsong Lv, and Nan Guan. Response time bounds for DAG tasks with arbitrary intra-task priority assignment. In *Proc. of ECRTS 2021*, 2021.

[14] Micaela Verucchi, Mirco Theile, Marco Caccamo, and Marko Bertogna. Latency-aware generation of single-rate DAGs from multi-rate task sets. In *Proc. of RTAS*, 2020.

[15] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *Proc. of Workshop on CODES/CASHE*, 1998.

[16] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. Random graph generation for scheduling simulations. In *Proc. of SIMUTools*, 2010.

[17] Penghao Sun, Zehua Guo, Junchao Wang, Junfei Li, Julong Lan, and Yuxiang Hu. Deepweave: Accelerating job completion time with deep reinforcement learning-based coflow scheduling. In *Proc. of IJCAI*, 2021.

[18] Chun-Hsian Huang. HDA: Hierarchical and dependency-aware task mapping for network-on-chip based embedded systems. *JSA*, 2020.

[19] Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle Puaut. Hiding communication delays in contention-free execution for SPM-based multi-core architectures. In *Proc. of ECRTS*, 2019.

[20] Kun Cao, Junlong Zhou, Peijin Cong, Liying Li, Tongquan Wei, Mingsong Chen, Shiyan Hu, and Xiaobo Sharon Hu. Affinity-driven modeling and scheduling for makespan optimization in heterogeneous multiprocessor systems. *TCAD*, 2018.

[21] Liu Shaoshan, Yu Bo, Guan Nan, Dong Zheng, and Akesson Benny. Industry challenge. In *Proc. of Industry Session (part of RTSS)*, 2021.

[22] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication centric design in complex automotive embedded systems. In *Proc. of ECRTS*, 2017.

[23] Mario Günzel, Niklas Ueter, and Jian-Jia Chen. Suspension-aware fixed-priority schedulability test with arbitrary deadlines and arrival curves. In *Proc. of RTSS*, 2021.

[24] Niklas Ueter, Mario Günzel, Georg von der Brüggen, and Jian-Jia Chen. Hard real-time stationary gang-scheduling. In *Proc. of ECRTS*, 2021.

[25] Youngeun Cho, Dongmin Shin, Jaeseung Park, and Chang-Gun Lee. Conditionally optimal parallelization of real-time DAG tasks for global EDF. In *Proc. of RTSS*, 2021.

[26] Suhail Nogd, Geoffrey Nelissen, Mitra Nasri, and Björn B Brandenburg. Response-time analysis for non-preemptive global scheduling with fifo spin locks. In *Proc. of RTSS*, 2020.

[27] Kecheng Yang and Zheng Dong. Mixed-criticality scheduling in compositional real-time systems with multiple budget estimates. In *Proc. of RTSS*, 2020.

[28] Sergey Voronov, Stephen Tang, Tanya Amert, and James H Anderson. AI meets real-time: Addressing real-world complexities in graph response-time analysis. In *Proc. of RTSS*, 2021.

[29] Zheng Dong and Cong Liu. An efficient utilization-based test for scheduling hard real-time sporadic dag task systems on multiprocessors. In *Proc. of RTSS*, 2019.

[30] Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, and Wanli Chang. DAG scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In *Proc. of RTSS*, 2020.

[31] Atsushi Yano and Takuya Azumi. Work-in-progress: Reinforcement learning-based dag scheduling algorithm in clustered many-core platform. In *Proc. of RTSS*, 2021.

[32] Longxin Zhang, Liqian Zhou, and Ahmad Salah. Efficient scientific workflow scheduling for deadline-constrained parallel tasks in cloud computing environments. *J. Inf. Sci.*, 2020.

[33] Yue Tang, Zhiwei Feng, Nan Guan, Xu Jiang, Mingsong Lv, Qingxu Deng, and Wang Yi. Response time analysis and priority assignment of processing chains on ROS2 executors. In *Proc. of RTSS*, 2020.

[34] Hyunjong Choi, Mohsen Karimi, and Hyoseung Kim. Chain-based fixed-priority scheduling of loosely-dependent tasks. In *Proc. of ICCD*, 2020.

[35] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B Brandenburg. Response-time analysis of ROS 2 processing chains under reservation-based scheduling. In *Proc. of ECRTS*, 2019.

[36] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *Proc. of ICCPS*, 2018.

[37] Zhishan Guo, Ashikahmed Bhuiyan, Di Liu, Aamir Khan, Abusayeed Saifullah, and Nan Guan. Energy-efficient real-time scheduling of DAGs on clustered multi-core platforms. In *Proc. of RTAS*, 2019.

[38] Kunal Agrawal, Sanjoy Baruah, Zhishan Guo, Jing Li, and Sudharsan Vaidhun. Hard-real-time routing in probabilistic graphs to minimize expected delay. In *Proc. of RTSS*, 2020.

[39] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *J. Real-Time Syst.*, 2005.

[40] Muhammad Sulaiman, Zahid Halim, Muhammad Waqas, and Doğan Aydın. A hybrid list-based task scheduling scheme for heterogeneous computing. *J. of Supercomput.*, 2021.

[41] Saroja Subbaraj, Revathi Thiagarajan, and Madavan Rengaraj. Multi-objective league championship algorithm for real-time task scheduling. *J. Neural. Comput. Appl.*, 2020.

[42] Jing Liu, Kenli Li, Dakai Zhu, Jianjun Han, and Keqin Li. Minimizing cost of scheduling tasks on heterogeneous multicore embedded systems. *TECS*, 2016.

[43] Hafiz Fahad Sheikh and Ishfaq Ahmad. Sixteen heuristics for joint optimization of performance, energy, and temperature in allocating tasks to multi-cores. *TOPC*, 2016.

[44] DI George Amalarethinam and GJ Joyce Mary. DAGEN-a tool to generate arbitrary directed acyclic graphs used for multiprocessor scheduling. *IJRIC*, 2011.

[45] Mishra Ashish, Sharma Aditya, Verma Pranet, Abhijit R Asati, and Raju Kota Solomon. A modular approach to random task graph generation. *Indian J. Sci. Technol.*, 2016.

[46] Julius Roeder, Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. Energy-aware scheduling of multi-version tasks on heterogeneous real-time systems. In *Proc. of the SAC*, 2021.

[47] Mahdi Mohammadpour Fard, Mahmood Hasanloo, and Mehdi Kargahi. Analytical program power characterization for battery depletion-time estimation. *TECS*, 2021.

[48] Chu-ge Wu, Wei Li, Ling Wang, and Albert Y Zomaya. An evolutionary fuzzy scheduler for multi-objective resource allocation in fog computing. *J. Future Gener. Comput. Syst.*, 2021.

[49] Weslley N Costa, Lucas P Lima, and Otavio A de Lima Junior. Extracting method of packet dependence from noc simulation traces using association rule mining. *Analog Integrated Circuits and Signal Processing*, 2021.

[50] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *proc. of WATERS*, 2015.