Generating Utilization Vectors for the Systematic Evaluation of Schedulability Tests

David Griffin, Iain Bate, Robert Davis
Real-Time Systems Research Group, University of York, UK

RTSS 2020
Introducing the DRS Algorithm

- **Dirichlet-Rescale (DRS) algorithm**

  $$\mathbf{u} = \text{DRS}(n, U, \mathbf{u}^{\text{max}}, \mathbf{u}^{\text{min}})$$

- **Returns:**
  A vector of \(n\) components (utilization values) \(\mathbf{u} = (U_1, U_2, \ldots, U_n)\) such that
  \[
  \sum_{i=1}^{n} U_i = U
  \]
  \[
  \forall i \quad U_i^{\text{max}} \geq U_i \geq U_i^{\text{min}} \geq 0
  \]

- **Inputs:**
  - \(n\) – size of the vector required
  - \(U\) – total utilization required
  - \(\mathbf{u}^{\text{max}} = (U_1^{\text{max}}, U_2^{\text{max}}, \ldots, U_n^{\text{max}})\) vector of maximum constraints
  - \(\mathbf{u}^{\text{min}} = (U_1^{\text{min}}, U_2^{\text{min}}, \ldots, U_n^{\text{min}})\) vector of minimum constraints
Motivation

Systematic evaluation of the effectiveness of schedulability tests

Supported by

Generation of synthetic task sets with a variety of different parameters

Supported by

Generation of unbiased utilization vectors compliant with constraints

Foundational layer is the focus of this work
Key criteria for utilization vector generation

- **Uniformity**
  - The vectors of utilization values generated must be unbiased i.e. the vectors must be uniformly distributed within the valid region
    - Bias in the sets of vectors generated can undermine the conclusions drawn from studies into schedulability test effectiveness (Bini and Buttazzo, 2005 [6])

- **Efficiency**
  - Necessary to generate millions of task sets to achieve statistically significant sample sizes in wide-ranging systematic evaluations
    - Typically 1000 task sets per data point for high quality results (Davis, 2016 [11])

- **Flexibility**
  - Capable of handling constraints on individual task utilization values
    - So the utilization vectors can be tailored to the specific requirements of the problem at hand (examples later), while still producing a uniform distribution of vectors within the valid region given by the constraints
Vectors and Simplices

- n-dimensional vectors with components that sum to $U$
- Each vector represents a point in n-dimensional space (n=3 for visualization)
- Canonical form $x+y+z=1$ with $x \geq 0, y \geq 0, z \geq 0$
- Equation $x+y+z=1$ defines a hyperplane (plane in 3-D space)
- Combined with inequalities defines a standard n-1 dimensional simplex embedded in n dimensional space (triangle in 3-D space)
- Vectors required are points uniformly distributed within this simplex
Mathematical background

- **Adding constraints**
  - Maximum constraints form a *constraints simplex* on the same hyperplane as the standard simplex \((x+y+z=1 \text{ and } x \leq 0.5, y \leq 0.45, z \leq 0.7)\)
  - Vectors required are points uniformly distributed within the *valid region* i.e. within the intersection of the constraints simplex and the standard simplex
  - **Duality** between the two simplices – we could generate points in either simplex and use the other as the constraints
  - Minimum constraints can be handled by transforming the problem into a canonical form where all minimum constraints are zero (see the paper)
Related work

- **UUnifast algorithm**
  - First work on this topic published in the Real Time Systems literature
  - Bini and Buttazzo, 2005 [6]
  - Solves the problem with no maximum or minimum constraints
  - Useful for single processor systems

- **Flat Dirichlet distribution**
  - Can also be used to solve the problem with no constraints for single processor systems
Related work

- **UUnifast-Discard algorithm**
  - Davis and Burns (2010) [14]
  - Developed for multiprocessor systems, where $U > 1$, but $U_i > 1$ is invalid
  - Addresses the problem of maximum (and minimum) constraints
  - Very simple (naïve) approach – uses UUnifast then discards any points that do not comply with the constraints
  - Suffers from the *curse of dimensionality*: If the constraints on each component halve the volume of the valid region then the proportion of useful points is $1/2^n$ (fine when $n=3$, not so good when $n=50$)
Related work

- **RandFixedSum**
  - Invented by Stafford, 2006 [28] and adapted for task set generation by Emberson et al., 2010 [17]
  - Efficiently addresses the problem of **symmetric** maximum and minimum constraints (i.e. the same constraints for all tasks)
  - De facto standard approach for modelling multiprocessor systems
  - Does not cater for **asymmetric** constraints and cannot be adapted to do so because of its reliance on symmetry for its efficiency
Dirichlet-Rescale (DRS) algorithm

**DRS Algorithm**

- Addresses the intractability drawbacks: of UUnifast-Discard (discarding points) and of RandFixedSum (would need to generate points in very many different simplices to deal with a valid region that is an irregular shape)

- Basic concept is to generate a point in the standard simplex then if it is not in the valid region, make a series of transformations shifting the coordinates of the point until it is within the valid region

- Crucially these transformations must preserve the uniform distribution of points
How DRS works

**DRS Algorithm outline operation**

1. Transform the problem into a canonical form by removing minimum constraints
2. Exploits duality to switch the standard and constraints simplices for efficiency
3. Generate a point $\mathbf{P}$ on the standard simplex using the Dirichlet distribution
4. If $\mathbf{P}$ satisfies the constraints then return $\mathbf{P}$ (reversing the initial transformation)
5. Otherwise, defines Simplex $\mathbf{S}$ based on the broken constraints ($\mathbf{S}$ contains $\mathbf{P}$)
6. Map Simplex $\mathbf{S}$ onto the standard simplex via a matrix transformation
7. This scale and translate transformation alters the coordinates of $\mathbf{P}$ making it more likely that the point will now be in the valid region
How DRS works

**DRS Algorithm outline operation**

1. Transform the problem into a canonical form by removing minimum constraints
2. Exploits duality to switch the standard and constraints simplices for efficiency
3. Generate a point $\mathbf{P}$ on the standard simplex using the Dirichlet distribution
4. If $\mathbf{P}$ satisfies the constraints then return $\mathbf{P}$ (reversing the initial transformation)
5. Otherwise, defines Simplex $\mathbf{S}$ based on the broken constraints ($\mathbf{S}$ contains $\mathbf{P}$)
6. Map Simplex $\mathbf{S}$ onto the standard simplex via a matrix transformation
7. This scale and translate transformation alters the coordinates of $\mathbf{P}$ making it more likely that the point will now be in the valid region
How DRS works

DRS Algorithm outline operation

1. Transform the problem into a canonical form by removing minimum constraints
2. Exploits duality to switch the standard and constraints simplices for efficiency
3. Generate a point $\mathbf{P}$ on the standard simplex using the Dirichlet distribution
4. If $\mathbf{P}$ satisfies the constraints then return $\mathbf{P}$ (reversing the initial transformation)
5. Otherwise, defines Simplex $\mathbf{S}$ based on the broken constraints ($\mathbf{S}$ contains $\mathbf{P}$)
6. Map Simplex $\mathbf{S}$ onto the standard simplex via a matrix transformation
7. This scale and translate transformation alters the coordinates of $\mathbf{P}$ making it more likely that the point will now be in the valid region
How DRS works

- **DRS Algorithm outline operation**
  1. Transform the problem into a canonical form by removing minimum constraints
  2. Exploits duality to switch the standard and constraints simplices for efficiency
  3. Generate a point \( P \) on the standard simplex using the Dirichlet distribution
  4. If \( P \) satisfies the constraints then return \( P \) (reversing the initial transformation)
  5. Otherwise, defines Simplex \( S \) based on the broken constraints (\( S \) contains \( P \))
  6. Map Simplex \( S \) onto the standard simplex via a matrix transformation
  7. This scale and translate transformation alters the coordinates of \( P \) making it more likely that the point will now be in the valid region
How DRS works

- Ensuring Uniformity
  - Distribution of initial points generated over the standard simplex is uniform
  - Hence the distribution of points is also uniform over Simplex $S$
  - The matrix transformation that maps Simplex $S$ onto the standard simplex is an **Affine** transformation (i.e. a scale and translate transformation).
  - Therefore the points that are uniformly distributed over Simplex $S$ become uniformly distributed over the standard simplex and hence uniformly distributed over the valid region
How DRS works (convergence)

- **Convergence**
  - Let $p = \frac{\text{volume(valid region)}}{\text{volume(standard simplex)}}$
  - After $q$ iterations, the minimum converged volume $c \geq 1 - (1 - p)^q$
  - As $q \to \infty$, $c \to 1$ and so the algorithm converges

- **Illustration of convergence**
  - Heat map color codes the number of rescales needed to converge:
  - Here $p = 0.25$ and all 1000 initial points converged within 24 rescales
  [Note this was done for illustration purposes with the duality optimization disabled, otherwise no rescaling would be necessary since every point generated would be within the smaller constraints simplex (blue)]
**DRS Performance**

- **Experiment A**
  - \( n = 50 \) and 10,000 runs for each \( U \) in \([0.05, 0.95]\) in steps of 0.05
  - For each run: \( \text{DRS}(n, U, u_{\text{max}}) \) with constraints \( u_{\text{max}} = \text{UUniform}(n, 1) \)

- **Number of Rescales (Box plot)**
  - Worst-case occurs for \( U = 0.5 \) when constraints and standard simplex are the same size
  - Max rescales < 200 (upper circle)
  - Min rescales (lower circle)
  - Mean (middle line of box)
  - Percentiles (5%, 25%, 75%, 95%)
DRS Performance

**Experiment B**
- Similar to Expt. A, but $U$ fixed at 0.5 and $n$ varied from 5 to 100 in steps of 5
- For each run: $\text{DRS}(n, U, u^{\text{max}})$ with constraints $u^{\text{max}} = U\text{Unifast}(n, 1)$

**Number of Rescales (Box plot)**
- Number of rescales gradually increases with increasing size of the vectors (number of tasks)
- Max rescales < 200 (upper circle)
- Min rescales (lower circle)
- Mean (middle line of box)
- Percentiles (5%, 25%, 75%, 95%)
Experiment B (continued)

Number of Retries

- Rescale operations can lead to the accumulation of Floating Point error
- A retry is done by generating another point if the total error (sum of component values minus required utilization) exceeds 0.01%
- Number of retries increases with increasing size of the vectors, but remains low for $n \leq 100$
DRS Performance

- **Experiment C**
  - Runtime to generate all the vectors needed for a standard “benchmark” schedulability analysis experiment (1000 vectors for each of 18 utilization levels from $U = 0.05$ to 0.95 in steps of 0.05, 18,000 vectors in all)

- **Runtimes**:
  - Used a Pi 4 to obtain reliable timings
  - Runtimes well approximated by a polynomial of order 3 (cubic function) $R^2 = 0.999$
  - Typical use would be on a laptop or desktop PC (e.g. Dell XPS 13 with Intel™ i7-1065G7 at 3.5GHz
    - ~6 seconds for 10 tasks
    - ~60 seconds for 50 tasks
    - ~6 minutes for 100 tasks
    (approx. 6 times faster than a Pi 4)
**DRS Performance**

- **Experiment D**
  - Verified the uniformity of the distribution of vectors produced by DRS via comparison with UUnifast-Discard

- **Statistical test:**
  - Examined the density of points produced in 1000 small reference simplices (within the valid region) via the DRS algorithm and UUnifast-Discard
  - Compared the Empirical Cumulative Distribution Functions (ECDF) using a statistical test: Kolmogorov-Smirnov (KS) test
  - KS-statistic = 0.04, p-value = 1.0
  - No evidence that the vectors produced come from different distributions
  - Cannot reject the null hypothesis that the distributions are the same

![ECDF Graph](image)
Main use is in the systematic evaluation of schedulability tests

- Used to underpin the generation of synthetic task sets with execution times derived from the utilization values

Asymmetric constraints:

- Occur when execution times have multiple values or are composed from multiple parts:
  - Mixed Criticality Systems (e.g. C(LO), C(HI))
  - Multi-core systems (e.g. processor demand, bus demand, memory demand, etc.),
  - Typical and worst-case execution times
  - Self-suspensions and resource locking

No constraints or symmetric constraints:

- DRS can be used to replace UUnifast for single processor systems, and RandFixedSum and UUnifast-Discard for multiprocessor systems
Mixed Criticality Systems Example

- **Schedulability Analysis Experiment**
  - Reproduced from the Adaptive Mixed Criticality (AMC) scheduling paper (Baruah et al., 2011 [4])

- **Using DRS:**
  - Independent control of total $U(LO)$ and $U(HI)$
  - Independent selection of $U_i(LO) \leq U_i(HI)$ and hence $C_i(LO) \leq C_i(HI)$
  - Eliminates generation of invalid (infeasible) task sets
  - $U_i(HI)$ generated by calling $\text{DRS}(n^{HI}, U_i^{HI}, u^1)$
  - Maximum constraints set to 1 for LO-criticality tasks and to $U_i(HI)$ for HI-criticality tasks
  - $U_i(LO)$ generated by calling $\text{DRS}(n, U^{LO}, u^{\text{max}})$
Mixed Criticality Systems Example

- **Schedulability Analysis Experiment**
  - Reproduced from AMC paper [4]
  - DRS highlights sharper transition of AMC and larger improvement over SMC
  - More nuanced and realistic results – could affect decisions on which methods to use

**Baruah et al.**

**DRS used in task set generation**
Why use an unbiased distribution of utilization vectors?

**What is meant by an unbiased?**
- Vectors generated are uniformly distributed across the valid region
- Does **not** mean the component values themselves are uniformly distributed (common misconception)

**Why use a unbiased distribution?**
- For generic schedulability analysis experiments, using a uniform distribution of utilization vectors means that each possible vector that complies with the constraints has the same chance of being selected
- The distribution is thus unbiased, provides full and fair coverage of all valid possibilities, and is therefore arguably the appropriate one to use
- Not using a uniform distribution of vectors risks biasing the results of schedulability analysis experiments
Easy ways of introducing bias…

1. **Confound variables** ($n$ and $U$)
   - Select $U_i$ from a uniform distribution [0,1] and keep adding tasks until the required total utilization is reached.
   - Confounds $n$ and $U$, so we cannot distinguish the effects of higher task set cardinality from those of higher task set utilization.

2. **Simple scaling** (UScale)
   - Select $n$ values for $U_i$ from a uniform distribution [0,1] and then scale them to achieve the required total utilization $U$.

3. **Addition of components** (UAdd)
   - Use UUnifast for each of multiple parts of $U_i$ and then add these values together.
Conclusion:
Why use the DRS algorithm?

- **Flexible - general purpose algorithm**
  - Supports asymmetric constraints on maximum and minimum utilization for each task
    - Used to obtain unbiased distributions when execution times have multiple values or are composed from multiple parts
    - Useful for tailoring task sets to specific problem requirements, limitations, or domain specific constraints
  - Can also be used to replace UUnifast, UUnifast-Discard, and RandFixedSum

- **High performance**
  - Supports efficient generation of task sets with cardinality up to $n = 100$ with individual constraints
  - Additional experiments show that DRS supports generation of task sets with cardinality up to $n = 200$ with a commensurate slowdown in performance

- **Python source code is publicly available**
  - Permanently archived at [https://doi.org/10.5281/zenodo.4118059](https://doi.org/10.5281/zenodo.4118059)
  - Can be installed via: `pip install drs` ([https://pypi.org/project/drs/](https://pypi.org/project/drs/))
  - Also provide a C library enabling the DRS algorithm in Python to be called directly from C/C++ code
Questions?